OCT 21, 1996

11573189

Nintendo 64

PROGRAMMING MANUAL

NU6-06-0030-001G of October 21, 1996

D.C.N. NU6-06-0030-001 REV G

"Confidential"

This document contains confidential and proprietary information of Nintendo and is also protected under the copyright laws of the United States and foreign countries. No part of this document may be released, distributed, transmitted or reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from Nintendo.

© 1995, 1996 Nintendo

1152189

Contents

List of Figures xvii

List of Tables xxi

PART I Getting Started

1. **Hardware and Software Installation Notes** 27
Hardware Installation 28
Software Installation 30
  READMEs and Release Notes 30
  Other Sources 30
  Executables 31

2. **Troubleshooting Software Bringup** 33
  Operating System 33
  Graphics 34
  Audio 36
  Integration 37
  Debugging CPU Faults 37

11573189

PART II Ultra 64 System Overview

- 3. **Hardware Architecture** 41
  - Execution Overview 42
  - RCP: Reality CoProcessor 43
  - RSP: Reality Signal Processor 44
  - RDP: Reality Display Processor 45
  - R4300 CPU 46
  - Memory Issues 47
  - Clock Speeds and Bus Bandwidth 48
  - Development Hardware 48
- 4. **Runtime Software Architecture** 51
  - Resource Access and Management 52
  - CPU Access 54
    - Message Passing Priority Scheduled Threads 54
    - CPU Data Cache 54
    - No Default Memory Management 55
    - Timers 55
    - Variable TLB Page Sizes 55
    - MIPS Coprocesser 0 Access 56
    - PI Manager 56
    - VI Manager 57
  - Memory Management 58
    - No Default Dynamic Memory Allocation 58
    - Region Library 58
    - Memory Buffer Placement 58
    - Memory Alignment 58
  - RCP Access and Management 60
  - Graphics Interface 61
    - Graphics Binary Interface 61
    - GBI Geometry and Attribute Hierarchy 61
    - GBI Feature Set 62
    - RSP Geometry Microcode 63

Audio Interface 64
RCP Task Management 65
    The "Simple" Example 65
GameShop Debugger 67
    WorkShop Debugger Heritage 67
    Debugger Components 67

5. Compile Time Overview 69

Database Modeling 70
    NinGen 70
    Alias 71
    Other Modeling Tools 71
    Custom Modeling Tools 71
Model to Render Space Database Conversion 72
    Existing Convertors 72
    Custom Convertors 72
    Conversion Considerations 72
Gamma Correction 74
Music Composition 75
Wavetable Construction 76
Building ROM Images 77
    C Compiler Suite 77
    ROM Image Packer 77
    Headers and Libraries 78
Host Side Functionality 79

11573189

PART III Ultra 64 Operating System

6. Operating System Overview 83
  - Overview 83
  - Threads 84
  - Messages 84
  - Events 85
  - Memory Management 85
  - Input and Output 86
  - Timers 87
  - Controller Pack File System 87
  - Debugging Support 87
  - Boot Procedure 87

7. Operating System Functionality 89
  - Overview 89
  - System Threads, Application Threads, and the Idle Thread 90
  - Thread Data Structure 90
  - Thread State 90
  - Scheduling and Preemption 91
  - Thread Functions 92
  - Exceptions and Interrupts 93
  - Events 94
  - Event and Interrupt Functions 96
  - Non-Maskable Interrupts and PRENMI 96
  - Internal OS Functions 98

115731 89

8. **Input/Output Functionality** 101
  Overview 101
  Design Approach 103
  Synchronous I/O vs. Asynchronous I/O 104
  Mutual Exclusion 105
  I/O Components 105
  System Exception Handler 106
  Device Manager 106
  Device-Dependent System Interface 108
9. **Basic Memory Management** 113
  Introduction 113
  Hardware Overview 113
  CPU Addressing 114
  Mixing CPU and SP Addresses 116
  Flushing the CPU Data Cache 118
  Clearing uninitialized data (Bss) section 119
  Physical Memory Allocation 119
10. **Advanced Memory Management** 121
  Introduction 121
  Mixing CPU and SP Data 121
  Using Overlays 122
  Using Multiple Waves 124
  Using the Region Allocation Routines 125
  Managing the Translation Lookaside Buffer 126

1153789

PART IV Ultra 64 Graphics

11. **Graphics Microcode** 131
- Microcode Functionality 132
  - gspFast3D 132
  - gspF3DNoN 132
  - gspLine3D 132
  - gspTurbo3D 132
  - gspSprite2D 133
  - gspSuper3D 133
- RSP to RDP command passing 134

12. **RSP Graphics Programming** 135
- RSP Overview 137
  - Display List Format 137
  - Segmented Memory and the RSP Memory Map 138
  - Interaction Between the RSP and R4300 Memory Caching 139
- Display List Processing 141
  - Connecting Display Lists 141
  - Branching Display Lists 142
  - Ending Display Lists 142
  - A Few Words about Optimal Display Lists 142
- Matrix State 144
  - Insert a Matrix 145
  - Pop a Matrix 145
  - Perspective Normalization 145
  - Note on Coordinate Systems and Big Numbers 146
  - A Few Words About Matrix Precision 147
- Vertex State 149
- Texture State 150
- Clipping and Culling 152

1153189

Vertex Lighting State 156
  RSP Microcode 156
  Normal Vector Normalization 157
  Ambient and Directional Lighting 157
  Specular Highlights 161
  Reflection Mapping 165
Vertex Fog State 169
Primitives 171
Controlling the RDP State 174

13. **RDP Programming 175**
  RDP Pipeline Blocks 176
  One-Cycle-per-Pixel Mode 177
  Two-Cycles-per-Pixel Mode 178
  Fill Mode 180
  Copy Mode 180
RDP Global State 181
  Cycle Type 181
  Synchronization 181
  Span Buffer Coherency 182
RS: Rasterizer 184
  Scissoring 184
TX: Texture Engine 186
  Texture Tiles 186
  Multiple Tile Textures 187
  Texture Image Types and Format 188
  Texture Loading 188
  Color-Indexed Textures 190
  Texture-Sampling Modes 191
  Synchronization 192
TF: Texture Filter 193
  Filter Types 193
  Color Space Conversion 194

1157389

CC: Color Combiner 195
  Color and Alpha Combiner Inputs Sources 195
  CC Internal Color Registers 197
  One-Cycle Mode 198
  Two-Cycle Mode 200
  Custom Modes 200
  Chroma Key 201
BL: Blender 203
  Surface Types 203
  Antialiasing Modes 204
  BL Internal Color Registers 205
  Alpha Compare 205
  Using Fog 206
  Depth Source 208
MI: Memory Interface 210
  Image Location and Format 210
  Fill Color 211
  Dithering 211

14. Texture Mapping 213
Graphics Binary Interface for Texture 216
  Primitive Commands 216
  Tile Related Commands 216
  Load Commands 216
  Sync Commands 216
  Mode Commands 216
Example Display List 218
Texture Image Space 219

1154318 9

Tile Attributes 221
  Format 221
  Size 221
  Line 222
  Tmem Address 222
  Palette 222
  Mirror Enable S,T 222
  Mask S,T 223
  Shift S,T 223
  SL,TL 224
  SH,TH 224
  Clamp S,T 224
Tile Descriptor Loading 225
Texture Pipeline 226
Tile Selection 228
  Functionality 228
  LOD Disabled 228
  LOD Enabled 229
  MIP Mapping 232
  Magnification 233
Texture Memory 239
  Memory Organization 239
  Texel Formatting 247
Texture Loading 248
Examples 255
Restrictions 259
  Texture Types and Modes 259
  Alignment 259
  Tiles 260
  Coordinate Range 260
Applications 261
  Multiple Tile Effects 261
Appendix A: LoadBlock Line Limits 264

115731789

15. Texture Rectangles (Hardware Sprites) 269
- Sampling Overview 271
- Simple Texture Effects 279
- Texture Types 288
- Multi-Tile Effects 292
- Tiling Large Images 297
- Color Index Frame Buffer 298
- Z-Buffering Texture Rectangles 299

16. Antialiasing and Blending 301
- Antialiasing 302
- Coverage Unit 306
- Z Stepper 308
- Blender 310
  - Color Blend Hardware 310
  - Fog 313
  - Coverage Calculation 314
  - Alpha Compare Calculation 315
  - Blender ADD Mode 317
  - Color Image Format 318
  - Image Alignment Requirements 320
  - Z Calculation 320
  - Z Image Format 322
  - Z Accuracy 325
- Video Filter 326

11513189

- Blender Modes and Assumptions 327
  - Opaque Surface Antialiased Z-Buffer Algorithm, OPA_SURF 327
  - Transparent Surfaces, XLU_SURF 329
  - Transparent Lines, XLU_LINE 331
  - Texture Edge Mode, TEX_EDGE 332
  - Decal Surfaces, OPA_DECAL, XLU_DECAL 333
  - Decal Lines, DEC_LINE 334
  - Interpenetration, OPA_INTER, XLU_INTER 335
  - Particle System Mode, PCL_SURF 336
  - Blender Modes Truth Table 337
  - Creating New Blender Modes 345
  - Visualizing Coverage 346

17. Sprites 349

- Application Program Interface (API) 351
  - Making Sprites 351
  - Manipulating Sprites 351
  - Drawing Sprites 353
- Data Structures and Attributes 354
  - Bitmap Structure 354
  - Sprite Structure 354
  - Attributes 355
- Tricks and Techniques 358
  - Sparse Sprites 358
  - Early-Ending Sprites 358
  - Variable Size Bitmaps 358
  - Explosions 358
  - Bitmap Re-use 358
  - Sprite Re-use 359
- Examples 360
  - Backgrounds 360
  - Text (Fonts) 360
  - Simple Game 360

1157389

18. **Sprite Microcode** 361
- Sprite Microcode Functionality 362
- Sprite Microcode API 363

PART V Ultra 64 Audio

19. **The Audio Library** 369
- Generating Audio Output 372
- Sampled Sound Playback 373
  - Representing Sound 373
  - Playing Sounds 373
- Sequenced Sound Playback 376
  - Representing the Sequence 376
  - Representing Instruments 377
  - Playing Sequences 378
  - Loops in Sequence Players 380
  - Controllers in Sequence Players 381
- The Synthesis Driver 382
  - Initializing the Driver 382
  - Building and Executing Command Lists 383
  - Synthesis Driver Sound Data Callbacks 383
  - Assigning Players to the Driver 384
  - Allocating and Controlling Voices 384
  - Effects and Effect Busses 385
  - Creating Your Own Effects 386
  - Parameter Description 388
  - Summary of Driver Functions 393
- Writing Your Own Player 394
  - Initializing the Player 394
  - Implementing a Voice Handler 395
- Implementing Vibrato and Tremolo 397

1157389

20. Audio Tools 401

- The Instrument Compiler: ic 402
    - Invoking ic 402
    - Writing ic Source Files 403
- The ADPCM Tools: tabledesign, vadpcm_enc, vadpcm_dec 412
    - tabledesign 412
    - vadpcm_enc 413
    - vadpcm_dec 414
- The MIDI File Tools: midicvt, midiprint & midicomp 416
    - midicvt 416
    - midiprint 416
    - midicomp 417
- Midi Receiving with Midi Daemon: midiDmon 419
- Instrument Editor 420
- Midi and the Indy 421
- The sbc Tool 423
    - sbc 423

21. Audio File Formats 425

- Bank Files 426
    - ALBankFile 426
    - ALBank 427
    - ALInstrument 428
    - ALSound 429
    - ALEnvelope 430
    - ALKeyMap 431
    - ALWavetable 432
- ADPCM AIFC Format 435
- Sequence Banks 438
- Compressed Midi File Format 439

11531389

22. Nintendo 64 Audio Memory Usage 441
Overview of audio RDRAM usage 442
    Audio Buffers 442
    Sample Rate, Frame Rate, and Other Factors 443
Optimizing Buffer Sizes 444
    Audio DMA Buffers 444
    Command List Size 446
    Output Buffer Size 446
    Audio Thread Stacksize 446
    Synthesizer Update Buffers and Sequencer Event Buffers 446
    The Audio Heap 447
    The Sequence Buffer 447
    The Bank Control File Buffer 447

23. Using The Audio Tools 449
Overview of Audio System 450
    Typical Development Process 451
    Common Values 452
Dealing With Constraints and Allocating Resources 453
    Determining Hardware Playback Rate 453
    Limits of Voices and Processing Time 454
    Division of Sounds and Music Into Banks 454
    Limits of ROM 454
Creating Samples 455

115313189

Playback Parameters and .inst Files 457
  Setting Sample Parameters in the .inst File 457
  Differences Between Sound Player and Sequence Player Use of .inst Files 457
  Envelopes 458
  Keymaps and Velocity Zones 458
  Tuning for Samples Recorded at the Hardware Playback Rate 459
  Tuning for Samples Recorded at Varying Rates 459
  Sounds 461
  Instruments 461
  Banks 462
  Creating Bank Files 462
MIDI Files 463
  Loops in the sequences. 463
  Putting Things Together Into Makefiles 466
General MIDI and the Nintendo 64 467

24. **Scheduling Audio and Graphics** 469

Scheduling Issues 470
  Command List Generation 470
  Command List Processing 470
Using the Scheduler 472
  Creating the Scheduler: osCreateScheduler() 472
  Adding Clients to the Scheduler: osScAddClient() 472
  Creating Scheduler Tasks: The OSScTask Structure 473
  474
  Sending Tasks to the Scheduler: osScGetTaskQ() 476

11573189

PART VI Ultra 64 Development Tools

25. GameShop Debugger 479
    Hardware Environment 479
    Software Environment 479
    Rmon Theory of Operation 481
    Programming Model 482
    Using the Debugger 484

PART VII Ultra 64 Performance Tuning

26. Performance Tuning Guide 491
Data Reduction 492
    Game World Organization 492
    Hierarchical Culling 495
Geometry Tuning (gspFast3D - Precise Microcode) 496
    Vertex Grouping 496
    Pre Lighting 496
    Clipping and Lighting 496
    Kinds of Polygons 497
    Textures instead of Geometry 497
    Geometric Level of Detail 497
Geometry Tuning (Turbo Microcode) 498
Raster Tuning (Fillrate) 499
    Disable Atomic Primitives 499
    Partial Sorting for Z-Buffer 499
    No Z-Buffer 499
    No Antialiasing 501
    Reduced Aliasing 501
CPU Tuning 502
    Parallel Execution of the CPU and the RCP 502
    Sorting 502

PART VIII Index

115389

List of Figures

Figure 1-1 Nintendo 64 GIO Card 28
Figure 2-1 CPU KSEG0-3 Addresses 34
Figure 2-2 RSP Addresses 35
Figure 3-1 Nintendo 64 Hardware Block Diagram 42
Figure 3-2 Block Diagram of the RCP 44
Figure 3-3 Development System 49
Figure 4-1 Application Resources 53
Figure 4-2 I/O Access and Management Software Components 56
Figure 4-3 Graphics Pipeline 61
Figure 4-4 Graphics Binary Interface (GBI) of an Airplane 62
Figure 4-5 Debugger Components 67
Figure 6-1 Nintendo 64 System Kernel 83
Figure 8-1 Logical View of RCP Internal Major Devices and Interface Modules 103
Figure 8-2 Interactions Between I/O Components Servicing Simple I/O Request 106
Figure 8-3 Interaction Between I/O Components and a Shared Device 108
Figure 12-1 Nintendo 64 Graphics Pipeline 135
Figure 12-2 Perspective Normalization Calculation 146
Figure 13-1 One-Cycle Mode RDP Pipeline Configuration 177
Figure 13-2 Two Cycle Mode RDP Pipeline configuration 178
Figure 13-3 RS State and Input/Output 184
Figure 13-4 Scissor/Clipping/Screen Rectangles 185
Figure 13-5 TX State and Input/Output 186
Figure 13-6 Tile Descriptors and TMEM 187
Figure 13-7 CI TMEM Partition 190
Figure 13-8 Texture Filter State and Input/Output 193

115789

Figure 13-9 Color Combiner State and Input/Output 195
Figure 13-10 RGB Color Combiner Input Selection 196
Figure 13-11 Alpha Combiner Input Selection 197
Figure 13-12 Chroma Key Equations 201
Figure 13-13 Blender State and Input/Output 203
Figure 13-14 Surface Types 203
Figure 13-15 Memory Interface State and Input/Output 210
Figure 13-16 Color and Z Image Pixel Format 210
Figure 13-17 Fill Color Register LSB Replication 211
Figure 14-1 Texture Unit Block Diagram 214
Figure 14-2 Image Space and Tile Space 219
Figure 14-3 Texture Pipeline 226
Figure 14-4 Texture Pipeline, contd 227
Figure 14-5 MIP Map Tile Descriptors 232
Figure 14-6 Magnification Interval Relative to LOD 233
Figure 14-7 MIP Map With Detail Texture Tile Descriptors 235
Figure 14-8 Sharpen Extrapolation 238
Figure 14-9 Physical Tmem Diagram 239
Figure 14-10 Tmem Loading 240
Figure 14-11 Four-Bit Texel Layout in Tmem 241
Figure 14-12 Eight-Bit Texel Layout in Tmem 241
Figure 14-13 Sixteen-Bit Texel Layout in Tmem 242
Figure 14-14 YUV Texel Layout in Tmem 243
Figure 14-15 Thirty-Two Bit RGBA Texel Layout in Tmem 243
Figure 14-16 Tmem Organization for Eight-Bit Color Index Textures 245
Figure 14-17 Tmem Organization for Four-Bit CI textures 246
Figure 14-18 Texel Formats in DRAM 249
Figure 14-19 Example of LoadTile Command Parameters 250
Figure 14-20 Wrapping a Large Texture Using Two Tiles 251
Figure 14-21 Wrapping a Large Texture Using One Tile 252
Figure 14-22 Example of LoadBlock Command Parameters 253
Figure 14-23 Wrapping, Mirroring, and Clamping 256

Figure 14-24	Wrapping Within a Texture Tile 257
Figure 14-25	Example of Texture Decals 258
Figure 15-1	Texture Rectangle Definition 270
Figure 15-2	Aliasing in a Sampled Image 271
Figure 15-3	Point Sampling Scaling Problem 272
Figure 15-4	Bilinear Filtering 274
Figure 15-5	Triangular Filtering 275
Figure 15-6	Copy Mode 277
Figure 15-7	Flipping Texture Rectangles 279
Figure 15-8	TextureRectangleFlip Command 281
Figure 15-9	Mirrored Tree 281
Figure 15-10	Wrapping on Several Boundaries of the Same Texture 282
Figure 15-11	Wrapped and Mirrored Tree 283
Figure 15-12	Effect of Changing SL, TL 284
Figure 15-13	Biasing Texture Coordinates for Positive SL, TL 285
Figure 15-14	Texture Billboard 287
Figure 15-15	Shrinking a Sprite 294
Figure 15-16	Texture Decals 296
Figure 15-17	Modulation 296
Figure 16-1	Edge With and Without Antialiasing 302
Figure 16-2	Unweighted Area Sampling 303
Figure 16-3	Antialiasing Data Flow 304
Figure 16-4	Coverage Calculation 306
Figure 16-5	Complementary Edges 307
Figure 16-6	Z-Buffer Planes 308
Figure 16-7	Subpixel Correction of Z 309
Figure 16-8	Alpha Compare in Copy Mode for 8-bit Framebuffer 316
Figure 16-9	Alpha Compare in One/Two-Cycle Mode 317
Figure 16-10	Hidden Bits 319
Figure 16-11	Color Image Formats 320
Figure 16-12	Z Encoding 322
Figure 16-13	Z Memory Format 324
Figure 16-14	Z Worst-Case Error 325

1153769

Figure 19-1 Audio Software Architecture 370
Figure 19-2 Effects Primitives 387
Figure 19-3 A simple echo effect 390
Figure 19-4 A nested all-pass inside a comb effect 391
Figure 26-1 Fixed Size Grid Database Organization 492
Figure 26-2 Quadtrees 493
Figure 26-3 Portals Connectivity Visibility 494
Figure 26-4 Bounding Sphere Test 495

1157318 9

List of Tables

Table 4-1 GBI Feature Set 62
Table 7-1 94
Table 7-2 Events Defined for the Nintendo 64 System 95
Table 9-1 32 Bit Kernel Mode Addressing 114
Table 12-1 gsSPDisplayList(Gfx *dl) 141
Table 12-2 gsSPBranchList(Gfx *dl) 142
Table 12-3 gsSPEndDisplayList(void) 142
Table 12-4 gsSPMatrix(Mtx *m, unsigned int p) 145
Table 12-5 gsSPPopMatrix(unsigned int n) 145
Table 12-6 gsSPPerspNormalize(unsigned short int s) 146
Table 12-7 gsSPVertex(Vtx *v, unsigned int n, unsigned int v0) 149
Table 12-8 gsSPTexture(int s, int t, int levels, int tile, int on) 150
Table 12-9 gsSPSetGeometryMode(unsigned int n) 154
Table 12-10 gsSPClearGeometryMode(unsigned int n) 154
Table 12-11 gsSP1Triangle(int v0, int v1, int v2, int flag) 171
Table 12-12 gsSPLine3D(int v0, int v1, int flag) 171
Table 12-13 gsDPFillRectangle(unsigned int ulx, unsigned int uly, unsigned int lrx, unsigned int lry) 172
Table 12-14 gsSPTextureRectangle(unsigned int ulx, unsigned int uly, unsigned int lrx, unsigned int lry, int tile, short int s, short int t, short int dsdx, short int dtdy) 172
Table 12-15 gsSPTextureRectangleFlip(unsigned int ulx, unsigned int uly, unsigned int lrx, unsigned int lry, int tile, short int s, short int t, short int dtdx, short int dsdy) 173
Table 13-1 Cycle Types 175
Table 13-2 Basic Operations of RDP Subblocks 176
Table 13-3RDP Pipeline Block Functionality in One-Cycle Mode 177
Table 13-4RDP Pipeline Block Functionality for Two-Cycle Mode 178

Table 13-5gsDPSetCycleType(type) 181

Table 13-6gsDPPipeSync() 181

Table 13-7gsDPFullSync() 182

Table 13-8gsDPPipelineMode(mode) 183

Table 13-9gsDPSetScissor(ulx, uly, lrx, lry) 185

Table 13-10 Texture Format and Sizes 188

Table 13-11gsDPLoadTextureTile(timg, fmt, siz, width, height, uls, ult, lrs, lrt, pal, cms, cmt, masks, maskt, shifts, shiftt) 189

Table 13-12gsDPLoadTextureTile_4b(pkt, timg, fmt, width, height, uls, ult, lrs, lrt, pal, cms, cmt, masks, maskt, shifts, shiftt) 189

Table 13-13gsLoadTLUT(count, tmemaddr, dramaddr) 191

Table 13-14gsDPSetTexturePersp(mode) 191

Table 13-15gsDPSetTextureDetail(mode) 192

Table 13-16gsDPSetTextureLOD(mode) 192

Table 13-17gsSetTextureLUT(type) 192

Table 13-18gsSetTextureFilter(type) 194

Table 13-19gsSetTextureConvert(mode) 194

Table 13-20gsSetConvert(k0,k1,k2,k3,k4,k5) 194

Table 13-21gsSetPrimColor(minlevel, frac, r, g, b, a), gsDPSetEnvColor(r, g, b, a) 198

Table 13-22 One-Cycle Mode Using gsDPSetCombineMode(mode1, mode2) 198

Table 13-23 Two-Cycle Mode Using gsDPSetCombineMode(mode1, mode2) 200

Table 13-24 One-Cycle Mode gsDPSetRenderMode(mode1, mode2) 204

Table 13-25 Two-Cycle Mode gsDPSetRenderMode(mode1, mode2) 205

Table 13-26gsDPSetFogColor(r, g, b, a) gsDPSetBlendColor(r, g, b, a) 205

Table 13-27gsDPSetAlphaCompare(mode) 206

Table 13-28gsSetFillColor(data32bits) NEED READABLE TITLE FOR THIS! 211

Table 14-1 Tile Format Encodings 221

Table 14-2 221

Table 14-3 Shift Encoding 223

Table 14-4 Tile Descriptor Index Generation with LOD Disabled 228

Table 14-5 Example of Tile Address and LOD Index Relationship 230
Table 14-6 Generation of Tile Descriptor Index With LOD Enabled and Magnifying 231
Table 14-7 Generation of Tile Descriptor Index With LOD Enabled and Not Magnifying 231
Table 14-8 Maximum tile sizes in TMEM 240
Table 14-9 Texel Output Formatting 247
Table 14-10 Limits on Number of Lines for LoadBlock Command 264
Table 16-1 P and M Mux Inputs 310
Table 16-2 A Mux Inputs 311
Table 16-3 B Mux Inputs 311
Table 16-4 Fog Mux Controls 313
Table 16-5 Antialiased Z-buffered Rendering Modes, G_RM_AA_ZB 339
Table 16-6 Antialiased Non-Z-Buffered Rendering Modes, G_RM_AA 341
Table 16-7 Point-Sampled Z-Buffered Rendering Modes, G_RM_ZB 343
Table 16-8 Point-Sampled Non-Z-Buffered Rendering Modes, G_RM 345
Table 19-1 Sound Player Functions 375
Table 19-2 Sequence Functions 377
Table 19-3 Bank Functions 378
Table 19-4 Sequence Player Functions 379
Table 19-5 Synthesizer Functions 393
Table 20-1ic Command Line Options 403
Table 20-2tabledesign Command Line Options 413
Table 20-3vadpcm_enc Command Line Options 414
Table 20-4vadpcm_dec Command Line Options 415
Table 20-5midicvt Command Line Options 416
Table 20-6midiprint Command Line Options 417
Table 20-7midicomp Command Line Options 417
Table 21-1ALBankFile Structure 427
Table 21-2ALBank Structure 427
Table 21-3ALInstrument Structure 428
Table 21-4ALSound STructure 429
Table 21-5ALEnvelope Structure 430

1157139

Table 21-6ALKeyMap Structure 431
Table 21-7ALWavetable Structure 433
Table 21-8ALADPCMWaveInfo structure 433
Table 21-9ALRawWaveInfo structure 434
Table 21-10ALADPCMLoop structure 434
Table 21-11ALADPCMBook structure 434
Table 21-12ALRawLoop structure 434
Table 22-1 DMA Buffer Length. 445
Table 23-1 Tuning to hardware playback rates. 460
Table 24-1OSScTask structure fields 473
Table 24-2OSTask structure fields 474

115731869

PART

Getting Started

115731 89

11573189

Chapter 1

Hardware and Software Installation Notes

This chapter describes how to install the Nintendo 64 development board into a Silicon Graphics Indy workstation. It also describes how to install the Nintendo 64 development software and where the software components are. located

This chapter is not a complete installation guide. You must be familiar with the standard SGI software installation procedures and GIO board installation in an Indy workstation.

11573189

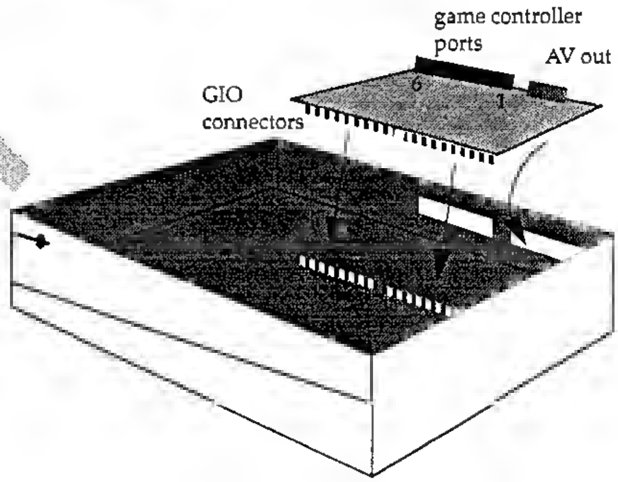
Hardware Installation

The Nintendo 64 Development Board is installed in the Indy workstation as described in the *Indy Workstation Owner's Guide* (see the chapter "Installing the GIO Option Board"). The following instructions supplement that chapter and serve as an errata. Figure 1-1 shows the placement of the Nintendo 64 Development board in the Indy workstation.

The board is secured in the workstation by four screws that attach it to the standoffs on the base board. When you install the board, be careful not to damage any jumper wires that may be present on the board.

The Nintendo 64 Development board is not supported by the *hinv* command. Once the board and software have been successfully installed, the boot monitor will echo "U64 Device found" during the power-up procedure. The application *ginv* in /usr/scr/PR/ginv can be used to print information about the installed development board such as the RCP version number, clock speed, and video mode.

Figure 1-1 Nintendo 64 GIO Card



The AV out port connector type is the same as that used on the current Super Nintendo Entertainment System. The cable that connects this port to an external television can be obtained from most stores that sell the SNES device. You can buy different cables to support Composite, S-Video RGB, or other formats that are standard in your country.

Note that the AV out can optionally be routed back to the Indy video input and audio inputs, allowing you to view and hear the gameboard on the local Indy workstation. The workstation accepts composite or S-video input as provided on separate SNES cables.

The game controller ports accept RJ-11 connectors (available on the U64 Development game controllers provided by Nintendo). There are connectors for six ports, though only connectors 1 through 4 are active. The connectors are named 1 through 6, and are numbered from left to right (when you view the connector from the back of the workstation). Plugging a controller into port 5 will cause the machine to hang.

1157369

Software Installation

The Nintendo 64 development software image is not the only software required for development. Your Indy workstation must also contain the following 5.3 products:

- dev
- c_dev
- compiler_dev
- gl_dev
- CaseVision, version 2.4
- WorkShop, version 2.4

Three products are bundled with the Nintendo 64 development software:

- GameShop
- ultra
- dmedia_eoe (version 5.5)

Note: Casevision and Workshop need to be installed before Gameshop. Workshop needs to be version 2.4 or earlier.

READMEs and Release Notes

After installation of Nintendo 64 development software, You will find a collection of sample demonstration applications in /usr/src/PR. A README_DEMOS file which describes each applications key features. You will also find the release notes in /usr/src/PR/relnotes. The release notes summerizes the differences from the last release and various bugs, workarounds and caveats of the system.

Other Sources

In /usr/src/PR/assets, you will find the source files for building the general MIDI bank. We created an initial complete general MIDI bank for testing purposes. For a game, we assume that you will gut the bank down to

11513189

including only those instrument and sounds that you need. Therefore, this directory gives you a starting point to do that.

In /usr/src/PR/libultra, you will find some pieces of the Nintendo 64 system library code (libultra.a). These are supplied to give a starting point on writing your own custom versions of these sub components. However, these sources require extensive SGI source tree build environment tools to actually build. Therefore, only the non buildable sources are shipped currently.

Executables

The first piece of software you will need to use is *gload*. This program downloads the ROM image onto the Nintendo 64 development board and starts execution. Soon after, you will need to use *dbgif* and *gvd* to debug your program.

- /usr/sbin/gload
- /usr/sbin/dbgif
- /usr/sbin/gvd

There are also conversion tools that help in converting data into Nintendo 64 format. For example, *flt2c* convertss a MultiGen database into a C data structure that can be compiled into binary form. Most of these tools reside in /usr/sbin but some are suppiled in source form in /usr/src/PR/conv. Keep in mind that these are templates for your own custom database conversion tools. We can not possibly address the need of all developers.

11573189

1157318 9

Chapter 2

Troubleshooting Software Bringup

1157318 9

This chapter describes common problems that you might encounter when you start bringing up your Nintendo 64 software. The potential problem areas are:

- operating system
- graphics
- audio
- integration

Operating System

Game locks up immediately.

A common error is to start the rmon thread at the same priority as the spawning thread. Rmon then immediately goes to sleep and locks up the system. The recommended way for starting the system is to create an idle thread in the boot procedure at a high priority. From the idle thread start all the other application threads, then lower the priority to zero and loop forever to become the idle thread. Note that the rmon thread is not needed for printfs. See the *osSyncPrintf (3P)* man page.

Game encounters a CPU exception.

During the development of your game, you may (intentionally or unintentionally) encounter various CPU exceptions (or faults) such as TLB miss, address error, or divide-by-zero. Currently, the system fault handler saves the context of the faulted thread, stops the faulted thread from execution, sends a message to any thread registered for the OS_EVENT_FAULT event, and dispatches the next runnable thread from the system run queue. If rmon is running, it would register for the OS_EVENT_FAULT event, receive the message from the exception handler, stop all user threads (except the idle thread), and send the faulted thread context to the host. If gload is running on the host, it would receive the faulted thread context and print its content to the screen. If gvd is running on the host, it would receive the fault notification and point you to where the fault occurred. If rmon is not running on the target, you probably experience a strange behavior (i.e. hang) in your game since the faulted thread can no longer run.

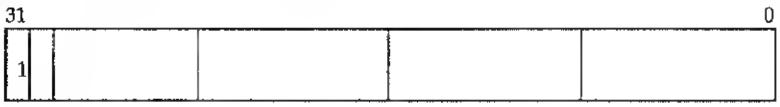
If you want to catch the OS_EVENT_FAULT event (instead of using rmon), you can use two internal OS functions to find the faulted thread and handle the exception yourself. They are *__osGetCurrFaultedThread (3P)* and *__osGetNextFaultedThread (3P)*. Please refer to their man pages for more information.

Graphics

There is no picture on the screen, but the drawing loop is running.

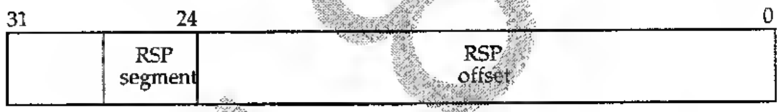
You are probably handing a bad segment address to the RSP graphics pipeline. This problem is easy to overlook, as there are no warnings. Make sure you thoroughly understand how a MIPS family processor performs addressing and how KSEG0 works (most games run in KSEG0). It allows cached access with no TLB translation. All CPU registers are accessible. KSEG addresses use the most significant bits of the address to indicate the addressing modes.

Figure 2-1 CPU KSEG0-3 Addresses



The RSP uses a segment addressing scheme with base pointers. It is very easy to hand a CPU KSEG0 address to the RSP by mistake and spend hours locating a simple error. Note that KSEG0 CPU address would reference a invalid segment if decoded as an RSP address.

Figure 2-2 RSP Addresses



For example, if you have the following code, the RSP/RDP pipeline will receive garbage:

```
Mtx matrix;
gSPMatrix(gdl++, &matrix, G_MTX_.....);
```

`matrix` is a KSEG0 CPU address `0x8xxxxxxx`. When this is handed to RSP, it fetches garbage. Below is a list of common commands with pointers:

- gDPSetColorImage
- gDPSetTextureImage
- gDPSetMaskImage
- gSPMatrix
- gSPViewport
- gSPVertex
- gSPDisplayList

Keep in mind that CPU addresses and RSP/RDP addresses uses different addressing schemes and are not interchangeable.

One useful way to debug possible display list problems is to link with the GBI dumping routines in libgu, and print out the display list. This will immediately show bad pointers and garbage matrices. See the man page for *guParseGbiDL (3P)* and *guParseRdpDL (3P)*.

Ending a Display List

Make sure that your recent gbi display edit has **gSPEndDisplayList** in each display list. Without this, the RSP will probably hang. The RDP requires a **gDPFullSync** at the end of the entire display list sequence to make the DP interrupt the CPU for notification.

Flaky Video

The beginning of the framebuffer and z-buffer addresses must be 64 byte aligned.

Audio

Alignment Issues

The audio system shares several data structures between the 4300 and the RCP. In order to avoid alignment problems, any buffer used by both the 4300 and the RCP should be allocated using the alHeapAlloc() routine. This will generate buffers with 16 byte alignment, avoiding all alignment issues as well as cache tearing issues.

Size and Number of buffers

A common error is to run out of buffers, particularly DMA buffers. Because the number of buffers needed is largely dependent on the music and sound effects used, it is not possible to provide guidelines. As music and sound effect complexity increases, the number of buffers needed will increase.

Audio Pops and Clicks

To avoid audio pops and clicks, all samples should start with at least one value of zero. Upon receiving a pre-nmi message it is important that the audio fade to zero output, or on subsequent bootup, there is a potential for a pop. If audio does not run at a high enough priority, the audio may not be generated before the previous buffer has completed. If this occurs there will be a period where no samples are played. This will usually generate a clear pop.

11543189

Integration

DMA Alignment

All DMA transactions in the Nintendo 64 must use 64 bit aligned for data in RDRAM. DMA transactions for data in ROM must use 16 bit aligned addresses.

Debugging CPU Faults

The "gdis" disassembler is a powerful debugging aide that can help you turn a cryptic crash dump (i.e the text that is printed in your gload window when your program takes an exception) into useful debugging information.

For example, you can disassemble the section named "code" (as specified in the spec file) in the "chrome" example application executable as follows:

```
% gdis -S -t .code.text letters
```

Here is a portion of the output ...

```
[ 144] 0x80200050:  27 bd ff 90       addiu     sp,sp,-112
[ 144] 0x80200054:  af bf 00 1c       sw        ra,28(sp)
  145:      int i, *pr;
  146:      char *ap;
  147:      u32 *argp;
  148:      u32 argbuf[16];
  149:
  150:      /* notice that you can't call rmonPrintf()
until you set
  151:       * up the rmon thread.
  152:       */
  153:
  154:      osInitialize();
[ 154] 0x80200058:  0c 08 04 c4       jal
osInitialize
[ 154] 0x8020005c:  00 00 00 00       nop
  155:
  156:      argp = (u32 *)RAMROM_APP_WRITE_ADDR;
[ 156] 0x80200060:  3c 0e 00 ff       lui       t6,0xff
[ 156] 0x80200064:  35 ce b0 00       ori
t6,t6,0xb000
```

```
[ 156] 0x80200068:  af ae 00 60       sw        t6,96(sp)
  157:      for (i=0; i<sizeof(argbuf)/4; i++, argp++) {
[ 157] 0x8020006c:  af a0 00 6c       sw
zero,108(sp)
  158:  osPiRawReadIo((u32)argp, &argbuf[i]); /* Assume
no DMA */
[ 158] 0x80200070:  8f af 00 6c       lw        t7,108(sp)
[ 158] 0x80200074:  8f a4 00 60       lw        a0,96(sp)
[ 158] 0x80200078:  27 b9 00 20       addiu     t9,sp,32
[ 158] 0x8020007c:  00 0f c0 80       sll       t8,t7,2
[ 158] 0x80200080:  0c 08 05 4c       jal
osPiRawReadIo
[ 158] 0x80200084:  03 19 28 21       addu      a1,t8,t9
[ 157] 0x80200088:  8f a8 00 6c       lw        t0,108(sp)
[ 157] 0x8020008c:  8f aa 00 60       lw        t2,96(sp)
[ 157] 0x80200090:  25 09 00 01       addiu     t1,t0,1
[ 157] 0x80200094:  2d 21 00 10       sltiu     at,t1,16
[ 157] 0x80200098:  25 4b 00 04       addiu     t3,t2,4
[ 157] 0x8020009c:  af ab 00 60       sw        t3,96(sp)
[ 157] 0x802000a0:  14 20 ff f3       bne
at,zero,0x80200070
[ 157] 0x802000a4:  af a9 00 6c       sw        t1,108(sp)
  159:      }

...
```

Notice that the C source is interleaved with the disassembled code, and that the PC is given in the second column.

When your program crashes, you can look up the error PC listed in the crash dump (it is identified as "epc") to determine where the program crashed and find the corresponding line in the source/disassembly listing.

11573189

PART

Ultra 64 System Overview

11573189

11573189

Chapter 3

Hardware Architecture

1151389

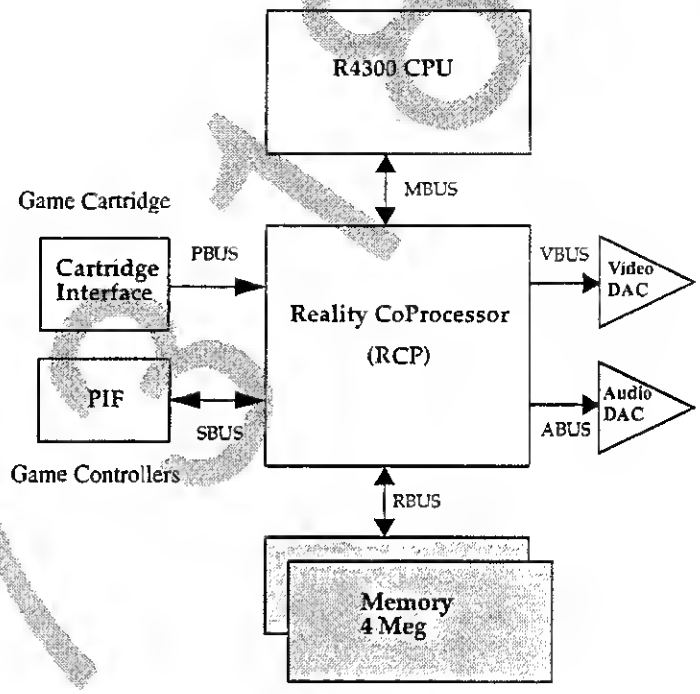
This chapter describes the hardware architecture of the Nintendo 64 game machine, in order to help you write software for the machine. Later sections of this manual describe the details you need to know to program each component.

The Nintendo 64 game consists of a number of hardware components that work together to produce the graphics and audio for the game. The heart of the system is the Reality CoProcessor (RCP). Attached to the RCP are memory chips, the MIPS R4300 CPU, and some miscellaneous I/O chips.

The RCP is the center of the game; all data must pass through it. It acts as the memory controller for the CPU. The RCP runs the graphics and audio microcode. The display portion of the RCP renders into the graphics framebuffer located in main memory. The video and audio portions of the RCP, DMA framebuffer, and audio data from main memory to drive the video and audio DACs. Figure 3-1, "Nintendo 64 Hardware Block Diagram," on page 42 is a block diagram of the Nintendo 64 system.

11573189

Figure 3-1 Nintendo 64 Hardware Block Diagram


R4300 CPU
MBUS
Game Cartridge
Cartridge Interface
PBUS
Reality CoProcessor (RCP)
VBUS
Video DAC
Audio DAC
ABUS
PIF
SBUS
Game Controllers
RBUS
Memory 4 Meg

Execution Overview

The CPU and RCP are both processors that can execute at the same time. *Threads* execute on the CPU and *tasks* execute on the RCP. Accesses to main memory from threads and tasks also occur in parallel.

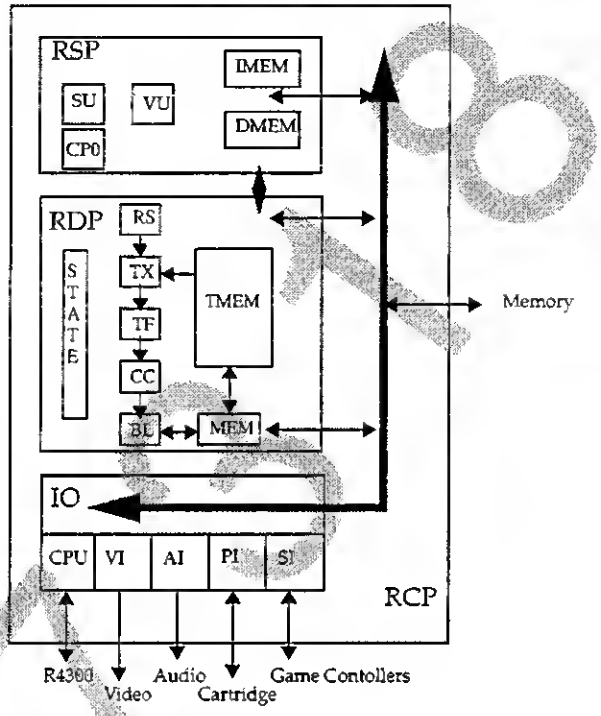
The game program runs on the R4300 CPU as a collection of threads, each of which has its own stack. The operating system is a collection of routines that can be called in a thread. The operating system controls which thread is running on the CPU. A thread can access all of physical memory. See Chapter 6, "Operating System Overview," for more information.

Tasks run on the RCP, which is a microcode engine that processes a *task list*. Task lists are generated by a thread running on the R4300 CPU and are stored in main memory. The game program creates the task list, calls an OS routine to load the appropriate microcode, and then starts the RCP running to process the task list. The microcode on the RCP reads the task list from main memory. The RCP task can also write into main memory.

RCP: Reality CoProcessor

The RCP is really a collection of processors, memory interfaces, and control logic. The Reality Signal Processor (RSP) is the microcode engine that executes audio and graphics tasks. The Reality Display Processor (RDP) is the graphics display pipeline that renders into the framebuffer. The memory interfaces provide access to main memory for the CPU, RSP, RDP, video interface, audio interface, peripherial devices, and serial game controllers. It is very important to remember that these interfaces may be active at the same time and that the RSP and RDP are running in parallel.

Figure 3-2 Block Diagram of the RCP


RSP
IMEM
SU
VU
DMEM
CP0
RDP
RS
S T A T E
TX
TMEM
TF
CC
BL
MEM
Memory
IO
CPU
VI
AI
PI
SI
RCP
R4300
Video
Audio
Cartridge
Game Contollers

RSP: Reality Signal Processor

The RSP is the processor used by the graphics and audio microcode. The RSP consists of a Scalar Unit (SU), a Vector Unit (VU), instruction memory (IMEM), and data memory (DMEM). The microcode is fetched from IMEM and has direct access to DMEM. The RSP can also access main memory using DMA. All memory references in the RSP are physical. However, the microcode uses a segment address table to translate segmented addresses provided in the task lists into physical addresses. The IMEM and DMEM are both 4 KB. The SU implements a subset of the R4000 instruction set. The VU has eight 16-bit elements.

11573189

For information on how the RSP is used to implement part of the graphics pipeline, see Chapter 12, "RSP Graphics Programming". Chapter 19, "The Audio Library," describes how the RSP is used in audio processing

RDP: Reality Display Processor

The RDP is the graphics display pipeline that executes an RDP display list generated by the RSP and CPU. The RDP consists of a Rasterizer (RS), a Texture Unit (TX), 4 KB of texture memory (TMEM), a Texture Filter Unit (TF), a Color Combiner (CC), a Blender (BL), and a Memory Interface (MI).

The RS rasterizes triangles and rectangles. The TX samples textures loaded in TMEM. The TF filters the texture samples. The CC combines and interpolates between two colors. The BL blends the resulting pixels with pixels in the framebuffer and performs z-buffer and anitaliasing operations. The MI performs the read, modify, and write operations for the individual pixels at either one pixel per clock or one pixel for every two clocks. The MI also has special modes for loading the TMEM, filling rectangles (fast clears), and copying multiple pixels from the TMEM into the framebuffer (sprites).

The RDP accesses main memory using physical addresses to load the internal TMEM, to read the framebuffer for blending, to read the z-buffer for depth comparison, and to write the z and framebuffers. The microcode on the RSP translates the segmented addresses in the task list into physical addresses.

The global state registers are used by all stages of the pipeline. There are a number of *sync* commands to provide synchronization. For example, a pipe sync is used before changing one of the rendering modes. This ensures that all previous rendering affected by the mode change occurs before the mode change.

The command list for the RDP usually comes directly from the RSP. However, it is possible to feed the RDP pipeline from a command list that has been stored in main memory.

See Chapter 13, "RDP Programming," for more information on the RDP.

1151319

Video Interface

The video interface reads the data out of the framebuffer in main memory and generates the composite, S-video, and RGB signals. The video interface also performs the second pass of the antialias algorithm. The video interface works in either NTSC or PAL mode, and can display 15- or 24-bit color pixels, with or without filtering, at both high and low resolutions. The video interface can also scale up a smaller image to fill the screen. For more information on how to set one of the 28 video modes and control the special features, see the man page for *osViSetMode (3P)*. Chapter 8, "Input/Output Functionality" also contains information on the video interface.

Audio Interface

The audio interface reads audio data out of main memory and generates the stereo audio signal. See Chapter 19, "The Audio Library" and Chapter 8, "Input/Output Functionality" for more information.

Parallel Interface

The parallel interface is the DMA engine that connects to the ROM cartridge. The PiManager thread is used to set up the actual DMA commands for all other threads. See Chapter 8, "Input/Output Functionality" for the list of PI functions.

Serial Interface

The serial interface connects the RCP with the game controllers through the PIF chip. To get the current state of the controllers, the application must send a command to query all the game controllers. The data will be available later. See Chapter 8, "Input/Output Functionality" for a list of all the controller functions.

R4300 CPU

The R4300 CPU is part of the MIPS R4000 family of processors. The R4300 consists of an execution unit with a 64-bit register file for integer and floating-point operations, a 16 KB instruction cache, an 8 KB writeback data cache, and a 32-entry TLB for virtual-to-physical address calculation. The

Nintendo 64 game runs in kernel mode with 32-bit addressing. 64-bit integer operations are available in this mode. However, the 32-bit C calling convention is used to maximize performace.

For more information on the R4300 and the operating system control of the CPU see the *MIPS Microprocessor R4000 User's Manual* and Chapter 6, "Operating System Overview".

Memory Issues

The main memory in the system is used in parallel by the R4300 CPU, the RSP microcode engine, the RDP graphics pipeline, and the other I/O interfaces of the RCP. The software is responsible for defining the memory map. See Chapter 9, "Basic Memory Management" for more details.

Addressing

The R4300 CPU can use physical or virtual addresses. The TLB maps virtual addresses into physical addresses. It is anticipated that programs will mainly use KSEG0 (cached, unmapped) addresses for instructions and data. The RSP hardware uses physical addresses. The microcode imposes a segmented addressing scheme to generate the physical addresses. Bits 24 through 27 of the segmented address are used to index into a 16-entry table to obtain the base address of the segment. The upper 4 bits are masked off. The lower bits are an offset into the segment. This scheme is used to create dynamic RSP task lists easily. The RDP hardware uses physical addresses. The RSP microcode translates the segmented addresses stored in the task list into physical addresses. The segment table in the RSP is initialized to all zeros. Every segment initially references memory starting at zero.

Data Cache

The R4300 CPU has an 8 KB writeback data cache. This means that when the CPU writes a variable, it may not be written to main memory until later. Since the RSP reads the task list directly from main memory, the dynamic portion of the task list must be flushed from the data cache before the RSP starts.

1153189

Take care in DMA operations also. The data buffer must be flushed from the cache before the write from memory occurs. The data buffer must be invalidated in the cache before a read into memory occurs. If the cache invalidate does not occur, a writeback from the cache may destroy data that has just been transfered into main memory by a read DMA. It is also a good idea to align I/O buffers on the 16-byte data cache line size, to avoid cache line tearing. Tearing occurs when a buffer and a unrelated variable share a cache line. The potential writeback of the variable could destroy data read into the I/O buffer.

Alignment

Note the various alignment restrictions:

- 8 byte alignment for most DMA
- 8 byte alignment for main memory, 2 byte alignement in ROM for PI
- 64 byte alignment for color framebuffers (cfb) and z-buffer
- 8 byte alignment for textures
- 16 byte alignment for vertices

Clock Speeds and Bus Bandwidth

Various system statistics and bandwidths:

- CPU - 94.0 Mhz
- RDRAM - 250 Mhz (9 bit bytes at 500 M/sec)
- RCP - 62.6 Mhz
- AI - variable, 3000-368000hz on NTSC, 3050-376000 on PAL
- VI - (depends on mode) NTSC, PAL, MPAL
- PI - 50 Meg/sec peak, 5 Meg/sec from typical slow ROMs
- SI - really slow HA HA No, Really...

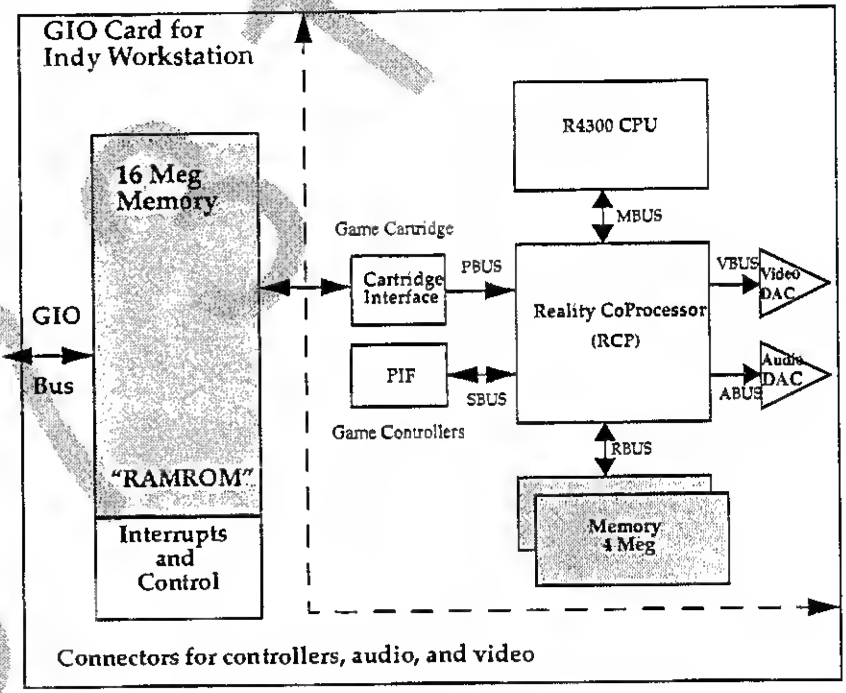
Development Hardware

The development system consists of an Nintendo 64 game card on a GIO card for the Indy workstation. The ROM cartridge is replaced by 16

1157319

megabytes of RAM, called the *ramrom*, that is accessible from both the Indy workstation over the GIO bus and the RCP over the PBUS. The workstation downloads the game software onto the GIO card and then the Nintendo 64 executes the game. The ramrom is also used to pass information by the debugger. The 4 Megabytes of main memory uses the 9 bit RDRAMs. The color and framebuffers can be placed anywhere in memory.

Figure 3-3 Development System


GIO Card for Indy Workstation
16 Meg Memory
"RAMROM"
Interrupts and Control
GIO Bus
Game Cartridge
Cartridge Interface
PBUS
PIF
SBUS
Game Controllers
R4300 CPU
MBUS
Reality CoProcessor (RCP)
VBUS
Video DAC
Audio DAC
ABUS
RBUS
Memory 4 Meg
Connectors for controllers, audio, and video

1157189

11573189

1153189

Chapter 4

Runtime Software Architecture

This chapter describes the runtime Nintendo 64 software architecture. It is intended as a brief tour of the overall architecture and discusses the basic design guidelines. More specific details are provided in subsequent chapters.

This chapter briefly covers the following topics:

- CPU: threads, messages, interrupts, cache coherency, tlbs
- IO: device library, device manager
- Memory: static allocation, region library
- RCP: tasks, command lists, yielding
- Graphics: graphics interface
- Audio: sequencer, audio player, driver, wavetable synthesis
- Application: typical application framework
- Debugger: debugger support for CPU and RSP

Resource Access and Management

The Nintendo 64 game machine is made up of a variety of resources. These resources include the CPU, memory, memory bus bandwidth, IO devices, the RSP, the RDP, and peripheral devices. The software is designed to provide raw access to all of the resources. The software layer basically translates logical functions and arguments into exact hardware register settings.

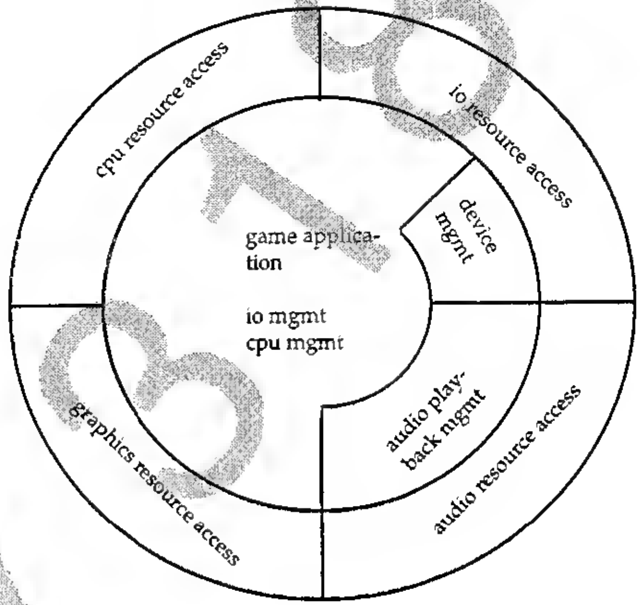
Management of most resources is left up to the game itself. Resources such as processor access and memory usage are too precious to waste by using some general management algorithm that is not tailored to a particular game's requirement. The only management layers provided are the audio playback and I/O device access.

The audio playback mechanism is fairly consistent from game to game. Only the sounds themselves are different. Therefore, a general tool to stream audio playback is useful. The I/O devices can be managed to provide simultaneous multiple access contexts for different threads. For example,

1157369

streaming audio data and paging in graphics database might require sharing access to the ROM.

Figure 4-1 Application Resources


cpu resource access
io resource access
device mgmt
game application
io mgmt
cpu mgmt
audio playback mgmt
audio resource access
graphics resource access

11573189

11513789

CPU Access

Message Passing Priority Scheduled Threads

To provide access to CPU compute cycles, Silicon Graphics provides a simple CPU scheduler to help the game manage multiple threads of control. These are the attributes of this scheduling scheme:

- Non-preemptive execution: The currently running thread will continue to run on the CPU until it wishes to yield. Preemption does occur if there is a need to service another, higher-priority thread awakened by an interrupt event. The interrupt service thread must not consume extensive CPU cycles. In other words, preemption is only caused by interrupts. Preemption can also occur explicitly with a yield, or implicitly while waiting to receive a message.
- Priority scheduling: A simple numerical priority determines which thread runs when a currently executing thread yields or an interrupt causes rescheduling.
- Message passing: Threads communicate with each other through messages. One thread writes a message into a queue for another thread to retrieve.
- Interrupt messages: An application can associate a message to a particular thread with an interrupt.

CPU Data Cache

The R4300 has a write back data cache to improve CPU performance. That means that when the CPU reads data, the cache may satisfy the read request eliminating the extra cycles needed to access main memory. When the CPU writes data, the data is written to the cache first and then flushed to main memory at some point in the future. Therefore, when CPU modifies data for the RCP's or IO DMA engine's consumption via memory, the software must perform explicit cache flushing. The application can choose to flush the entire cache or just a particular memory segment. If the cache is not flushed, the RCP or DMA may get stale data from main memory.

Before the RCP or IO DMA engines produce data for the CPU to process, the internal CPU caches must be explicitly invalidated. You don't want the CPU to be examining old stale data that is in the cache. The invalidation must occur before the RCP or DMA engine place the data in main memory. Otherwise, there is a chance that a write back of data in the cache will clobber the new data in main memory.

Since the software is responsible for cache coherency, keeping data regions on cache line boundaries is a good idea. A single cacheline containing multiple data produced by multiple processors can be difficult to keep coherent.

No Default Memory Management

As shown above, the Nintendo 64 operating system provides multi-threaded message-passing execution control. The operating system does not impose a default memory management model. It does provide a generic Translation Lookaside Buffer (TLB) access. The application can use the TLB to provide for a variety of operations such as virtual contiguous memory or memory protection. For example, an application can use TLBs to protect against stack overflows.

Timers

Simple timer facilities are provided, useful for performance profiling, real-time scheduling, or game timing. See the man page for *osGetTime (3P)* for more information.

Variable TLB Page Sizes

The R4300 also has variable translation lookaside buffer (TLB) page size capability. This can provide additional, useful functionality such as the "poorman's two-way set-associative cache," because the data cache is 8 KB of direct-mapped memory and TLB pages size can be set to 4 KB. The application can roll a 4 KB cache window through a contiguous chunk of memory without wiping out the other 4 KB in cache.

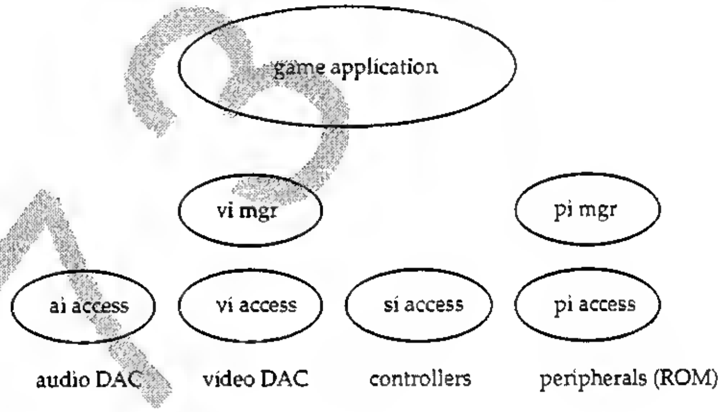
MIPS Coprocesser 0 Access

A set of application programming interfaces (APIs) are also provided for coprocessor 0 register access, including CPU cycle accurate timer, cause of exception, and status.

I/O Access and Management

The I/O subsystem provides functional access to the individual I/O hardware subcomponents. Most functions provide for logical translation to raw physical access to the I/O device.

Figure 4-2 I/O Access and Management Software Components


game application
vi mgr
pi mgr
ai access
vi access
si access
pi access
audio DAC
video DAC
controllers
peripherals (ROM)

PI Manager

Nintendo 64 also provides a peripheral interface (PI) device manager for multiple threads to access the peripheral device. For example, the audio thread may want to page in the next set of audio samples, while the graphics thread needs to page in a future database. The PI manager is a thread that waits for commands to be placed in a message queue. At the completion of the command, a message is sent to the thread that requested the DMA.

11573189

VI Manager

A simple video interface (VI) device manager keeps track of when vertical retrace and graphics rendering is complete. It also updates the proper video modes for the new video field. The VI manager can send a message to the game application on a vertical retrace. The game can use this to synchronize rendering the next frame.

115731 89

Memory Management

No Default Dynamic Memory Allocation

The Nintendo 64 software does not impose a memory map on the game. The Nintendo 64 system leaves the memory allocation problem up to the game application. It assumes that the application knows the memory partitioning scheme most suitable for the particular game. However, the Nintendo 64 library does have a heap library that is available.

Region Library

The Nintendo 64 system does provide a region allocation library that can partition a memory region specified by the application into a number of fixed-sized blocks. This gives the application the capability of using a dynamic memory allocation scheme. However, the game application must be able to handle the case when memory in the region has run out.

Memory Buffer Placement

There are some optimizations on the placement of memory buffers. For example, it is best to keep the color and depth buffers on separate 1 MB memory banks. The RDRAM has an active page register for each megabyte. Spliting the color and z-buffers into seperate megabytes, prevents the memory system from constantly having to change the page register. This technique minimizes page misses.

Memory Alignment

The DMA engines responsible for shuffling data around in the hardware all require the 64-bit aligned source address, the destination address, and lengths. Addresses in ROM do not have this 64 bit alignment restriction. ROM addresses only need to be 16-bit aligned. The loader from the compiler suite (see the man page for *ld (1)*) makes sure that all C-language `long long` types are 64-bit aligned.

1157189

Using C language, the stack for a thread must also be 64-bit aligned. Therefore, all stacks should be defined as `long long` and type-casted when calling *osCreateThread*. See the man page for more details.

115731 89

RCP Access and Management

The CPU has control over access to the RCP. The RSP and RDP portions of the RCP can be used individually, or as a group. The CPU creates a task list that specifies what microcode to run and what command list to execute. The task is then run on the RSP. There are OS commands to start the task and to yield (ie preempt) a task. The RDP usually receives graphics rendering commands directly from the RSP. However, it is also possible to drive the RDP from a list that is in DRAM.

1157319

11573789

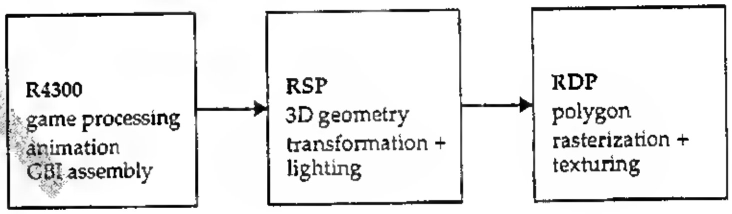
Graphics Interface

Nintendo 64 uses a display list hierarchy to describe what to render. 3D geometry transformation and rasterization are accelerated by RSP and RDP respectively. There is no immediate mode rendering. The R4300 CPU generates the display list in memory, then the RCP fetches the displaylist and renders the graphics.

Graphics Binary Interface

Nintendo 64 renders graphics using a display list interface called graphics binary interface (GBI). The CPU assembles the GBI structure in RDRAM for the RSP/RDP to render. The RSP must first be downloaded with graphics microcode to perform geometry transformation. The RDP performs polygon rasterization. RSP and RDP state machines are described in more detail in Chapter 12, "RSP Graphics Programming" and Chapter 13, "RDP Programming".

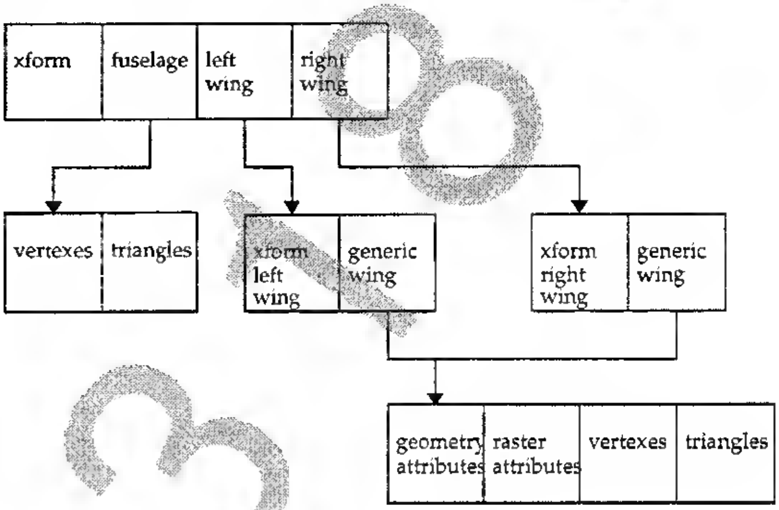
Figure 4-3 Graphics Pipeline



GBI Geometry and Attribute Hierarchy

The GBI structure describes a hierarchy of geometry and its attributes. This tree is traversed depth first and the graphics pipeline attributes are sequentially modified during traversal. Both geometry (RSP) and raster (RDP) attributes are contained in a GBI structure.

Figure 4-4 Graphics Binary Interface (GBI) of an Airplane


xform
fuselage
left wing
right wing
vertexes
triangles
xform left wing
generic wing
xform right wing
generic wing
geometry attributes
raster attributes
vertexes
triangles

1157389

GBI Feature Set

The graphics binary interface (GBI) contains many 3D graphics features. An algorithmic description of many of these features is in the *OpenGL Programmer's Guide*. Table 4-1, "GBI Feature Set," on page 62 lists the basic features of the GBI pipeline.

Table 4-1 GBI Feature Set

Processor	**Functionality**
CPU	GBI assembly

Table 4-1 GBI Feature Set

Processor	Functionality
RSP	matrix stack operations 3D transformations frustum clipping and back-face rejection lighting and reflection mapping polygon and line rasterization setup
RDP	polygon rasterization texturing/filtering blending z-buffering antialiasing

RSP Geometry Microcode

There are three different versions of RSP geometry microcode: gspFast3D, gspLine3D, and gspTurbo3D. The gspFast3D microcode is the optimized, full-featured 3D polygonal geometry microcode. The gspLine3D is the optimized, full-featured 3D line geometry microcode. The gspTurbo3D is the optimized, reduced-featured 3D polygonal geometry microcode. All of these microcode types come in two versions. One version of the microcode has the RSP output the rasterization and attribute commands directly to the RDP. The other version outputs RDP commands to DRAM. Writing the RDP commands to DRAM could be used to overlap graphics and audio. For example, you could use the RSP for audio processing while the RDP is processing commands stored in DRAM. Storing the RDP commands in DRAM may also be useful for debugging.

Audio Interface

Access to the audio subsystem is provided through the functions in the Audio Library. The Audio Library supports both sampled sound playback for sound effects and wavetable synthesis from MIDI files for background music. For more information on the Audio Library, please refer to Chapter 19, "The Audio Library".

115731 89

RCP Task Management

Both the audio and graphics libraries provide support for generating command lists to be executed on the RCP, but they do not handle the command list execution. It is therefore necessary for the application to manage the scheduling and execution of RCP tasks (command lists and microcode) on the RCP. To facilitate this, the development package includes an example RCP scheduler.

The "Simple" Example

The structure of the scheduler included with the "Simple" application is described briefly below. Please refer to the example code in the "Simple" directory for more details.

The Scheduler Thread

The scheduler thread is responsible for collecting display/command lists from other threads and assigning them to RCP tasks for scheduling and execution so that real-time constraints are met. This thread has the highest priority of the application threads, to insure that scheduling occurs periodically.

The scheduler executes task on the RCP based on the retrace interrupt and then monitors the progress, yielding the graphics tasks periodically to interleave audio tasks, if necessary.

Other Application Threads

The next highest priority application thread is the Audio Manager thread. It is responsible for creating audio display lists, sending them to the scheduler for execution, and transferring the finished audio to the codecs. It has a higher priority than the game thread, to prevent audio clicks caused when the audio thread can't meet its real-time constraints.

Note: The Audio Manager thread is essentially a low-level wrapper around the al**AudioFrame** call (see "The Synthesis Driver" on page 382 for details). Higher-level Audio Library calls are made from the game thread.

1151319

The game thread is responsible for generating graphics display lists and sending them to the scheduler for execution. In addition, the game thread handles the controller input, makes calls to the Audio Library, and performs other tasks traditionally found in the game's "main loop."

1157318

11573189

GameShop Debugger

WorkShop Debugger Heritage

The GameShop debugger (gvd) derived its heritage from the Silicon Graphics WorkShop application development tools. It is a source level windowing debugger environment that enables debugging of both the CPU and RSP software.

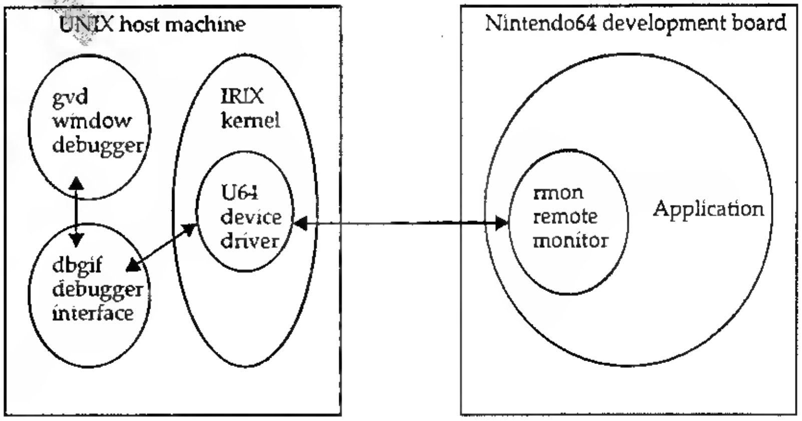
Debugger Components

The debugger is actually composed of several different components shown in Figure 4-5, "Debugger Components," on page 67

There are two debugging paths. The first path is a C source level windowing debugger, gvd, which has most of the features of common multi-threaded debuggers. It talks to dbgif, which interfaces to the rmon debug thread through the Nintendo 64 device driver in IRIX.

The second path is the popular printf traces within the application. rmonPrintf() display the messages in the shell that executed dbgif.

Figure 4-5 Debugger Components


UNIX host machine
gvd window debugger
dbgif debugger interface
IRIX kernel
U64 device driver
Nintendo64 development board
rmon remote monitor
Application

The rmon debugger thread is actually a high-priority thread in the game application and uses many operating system resources. Therefore, the debugger and rmonPrintf cannot be used to debug system-level code.

For information on using GameShop Debugger see Chapter 25, "GameShop Debugger."

11573189

Chapter 5

Compile Time Overview

1157189

This chapter describes the flow of tools required to go from 3D model design and music composition to cutting the actual ROM cartridge. In addition to the standard C compiler suite, the Nintendo 64 software release supplies a number of other tools particular to the Nintendo 64 software development environment. The source code to some of these tools is provided as an example to help you create your own customized tools that give your game an advantage in the game marketplace. This chapter includes the following sections:

- database modeling
- model space to render space database conversion
- music composition
- wavetable construction
- building ROM images
- host side functionality

Database Modeling

To do real-time 3D graphics, you need modeling tools to create geometry. Because many off-the-shelf modeling tools are available, there is no modeling package in the Nintendo 64 development kit from Silicon Graphics. Nintendo has contracted two top modeling package companies to provide the database modeling solution (MultiGen and Alias).

For texture-map images and traditional 2D sprite-type games, you may desire image conversion, editing, and paint software. These are not provided as part of the Nintendo 64 development kit.

All of the example applications and source code, including sample image conversion programs, use the popular SGI RGB image format. Additional related, but unsupported software, may be obtained from SGI via the 4Dgifts product, anonymous ftp via `sgi.com`, or from the user community on the internet (see `comp.graphics` or the `comp.sys.sgi` hierarchy). One of the more popular publicly available packages containing image conversion and manipulation software is PBMPLUS, widely available on the internet.

NinGen

NinGen is a 3D modeling package from MultiGen. It is a derivative of their traditional 3D modeling software, together with an Nintendo 64 database format convertor. The traditional key strength of MultiGen is their ability to provide 3D modeling tools for the real-time commercial and military flight/vehicle simulation market.

For this market, many database techniques developed for a real-time flight simulator are available in NinGen. Some basic features include:

- Geometric level of detail.
- Binary separating planes for depth-ordered rendering. This is required if you don't use the z-buffer.
- Many polygon count reduction tools. The goal is the best model with the lowest polygon count.

Alias

Historically, Alias has provided 3D animation and modeling tools for the computer-generated film and animation market segment. Beautiful models, sophisticated motion paths, and fast development time are all vital to success in this marketplace. Here is a sample of some of the strong features of the Alias software package:

- NURBs based modeler provides smooth surfaces on models.
- Motions paths and inverse kinematics give complex motion.
- Special effects such as particle systems, many different kinds of lights, and texturing capabilities improve picture quality.

Other Modeling Tools

Besides Alias and MultiGen, there are other modeling packages on the market. SoftImage and Nichimen Graphics are also traditional film and animation market tool suppliers. On the PC, the Autodesk 3DStudio is entering the animation market from the very low end of the price spectrum.

Film and animation tools have many features that can be extracted for real-time animation. Figuring out how to extract these special features out of theses tools can help you give your game application an advantage. For example, you might be able to use particle system tools to generate texture maps. Flipping this texture book on some morphing geometry to approximate the group motion of a system of particles. This may give you fire, water, and other interesting objects.

Custom Modeling Tools

For special game application requirements, you may need to create your own custom modeling packages. Obviously, it is time-consuming to build such a software package in house. The advantage, however, is that you can customize the databases to the requirements of your game. For example, you might be able to gain rendering display performance if you are able to give hints to your modeler about how to order geometry.

Model to Render Space Database Conversion

This section outlines issues you may face when converting from a modeling database to a rendering database.

Existing Convertors

Both NinGen and Alias software packages have database convertors to convert to the Nintendo 64 format (Graphics Binary Interface).

Custom Convertors

Some of you may want to write your own database convertors because you want to manage a certain resource or attribute in a different way, tailored to your game. Silicon Graphics provides a sample convertor, *flt2c*(1P), from the MultiGen flt file format to the Nintendo 64 format. In addition, Silicon Grapics provides a converter from the SGI IRIS image format to the Nintendo 64 texture memory format, *rgb2c*(1P). These sample convertors are not complete, nor are they designed to be totally efficient; they are just meant to be a template to help you understand what a convertor is and what it needs to do.

Conversion Considerations

There are many efficiency considerations to keep in mind when you are writing a database convertor. Here are a few:

- Redundant hierarchical transformations should be eliminated. Transformations should be used for articulated parts or instancing, not for preserving modeling hierarchy.
- Since the geometry transformation subsystem has a vertex cache, block loading 16 vertexes to render as many triangles as possible has better performance.
- On-chip texture memory is not large (4 KB). If you are stamping trees in your scene, you should render in texture order. Keep in mind that texture order may require a z-buffer, which requires additional dram

bandwidth. You may need to experiment to find the best trade-off for your game.

- The display pipeline has many attribute states. You may want to determine which sets are global and local to an object. Learn how to manage these attributes to best fit the kind of game you are creating.

115731189

Gamma Correction

The SNES and Super Famicom do not have gamma correction hardware but the Nintendo 64 does. Some developers have indicated that the colors on the Nintendo 64 look "washed out" with gamma correction turned on.

If you are currently writing games for SNES or Super Famicom (or any machine that does not have gamma correction), your production path is likely to be setup to compensate for the lack of gamma correction hardware. In other words, you are probably picking pre gamma corrected colors. If you use this same production path and turn Nintendo 64 gamma correction on, you will get the wash out effect because you would have gamma corrected twice.

To undo the first gamma correction, square and shift down by 8 each color component (assuming 8 bit color) or rework your path to exclude the gamma correction stop, leaving gamma correction to the hardware.

Every step in your production path must be involved in the color selection process: modeling/paint software, computer monitors, image conversion software, the game software, and the Nintendo 64 hardware.

Gamma correction on the Nintendo 64 is recommended; the antialiasing and video hardware work best when it is enabled.

1157389

Music Composition

Music composition involves the creation of midi sequences and then importing them into the game. Midi sequences can be created using any of a variety of sequencer applications. (Performer, Vision, Cubase, MasterTracks, to name a few) After the sequences are saved as Midi files, they should be converted before being included in the game. If you are planning to use the compact Midi sequence player, the sequences should be run through midicmp. If you are using the regular sequence player, the sequences are run through midicvt. After the sequences are converted, they can be assembled into sequence banks with the sbk tool. This is optional, midi sequences can be used without being part of a sequence bank. To actually include the sequences in the game, a segment containing the sequence data should added to the spec file. (See the demo app. simple for an example of this.)

For information on how to use sequences in a game see,Chapter 19, "The Audio Library,"

1157399

Wavetable Construction

The audio library can use either compressed or uncompressed wavetables for sound reproduction. In either case, the wavetables are first created using the digital recording/editing system of the sound designer's choice. The wavetables are then stored as AIFF files. If the samples are to be compressed, the first step is to produce a compression table using tabledesign. After the compression table has been built, the wavetable is compressed using vadpcm_enc. This will generate a type of AIFC file that is unique to the Nintendo. (Note that AIFC files created with other software tools are not compatible with the compression scheme used by the Nintendo.)

After the wavetables have been converted to AIFC files, (or left as AIFF files if no data compression is desired) they need to be assembled into banks so that the Audio Library can reference them correctly. To accomplish this, the sound designer must first create a .inst file, which is a text file that specifies the parameters for sound playback and the wavetable files. The .inst file is then used by ic to create the bank files. The bank files can then be included in the game by placing them in segments in the applications spec file. (The creation of .inst files and the use of ic is covered in detail in Chapter 20, "Audio Tools,")

1157389

Building ROM Images

A final set of tools, headers and libraries are available to pack your database and code into a final ROM images for the Nintendo 64. The Nintendo 64 development environment heavily leverages the C compiler and preprocessor tools to process symbolic data into binary objects. A ROM packing tool, *makerom*(1P) packs these objects into a single monolithic ROM image according to a specification of where these objects go.

C Compiler Suite

Currently, the Nintendo 64 development environment has only been verified with the IRIX 5.3 MIPS C-compiler suite. The interfaces provided do not rely on proprietary features of this compiler; however backend tools such as *makerom* may rely on specifics of the MIPS symbol table format.

It is required that all modules be compiled or assembled with the **`-non_shared`** and `-G 0` compilation flags; neither position independent code or a global data area is supported. Since the MIPS R4300 supports the MIPS II instruction set, the **`-mips2`** flag is also recommended, as well as optimization flags (`-O and -O3`).

ROM Image Packer

The ROM image packer (*makerom*) takes as input relocatable objects created by the compiler and performs the final relocations of code symbols. To perform these relocations, it invokes a next generation link editor that allows objects to be linked at arbitrary addresses specified by the developer. After these relocations, *makerom* extracts the code and initialized data portions of the resulting binary and packs them onto a ROM image. The *makerom* tool can also copy raw data files to the ROM as desired.

Note: When building a ROM image for the console (as opposed to the development system), be sure to

- link with libultra.a and not libultra_d.a
- remove all calls to printf and its variations from your application.

- remove any functions specific to the development board (such as command line parsing or logging) from your application.

Headers and Libraries

Although the Nintendo 64 API includes interfaces for a wide variety of areas, the interfaces are made available by including a single header file, */usr/include/ultra64.h*, and by linking with a single library, */usr/lib/libultra.a* (or */usr/lib/libultra_d.a*). The library routines are broken into their finest level of granularity, so applications "pay as they go", only including routines they actually use.

Note there are two versions of the Nintendo 64 library: a debug version (*/usr/lib/libultra_d.a*) and a non-debug version (*/usr/lib/libultra.a*). The debug version of the library provides additional run time checks at the expense of some space on the ROM and DRAM, as well as some performance. The kinds of checks performed include argument checking (especially hard to find alignment problems), improper use of interfaces, audio resource problems, etc. It is recommended that the debug library be used in initial development, and then replaced by the non-debug library later in the development cycle.

In case of error, the game loading program *gload*(1P) will interpret and display the errors on the host.

115739

Host Side Functionality

During development, it may be desirable to copy data to and from the Indy host to the game. For example, a MIDI sequence could be repeatedly edited on the host and them played on the Nintendo 64. Of course this could be accomplished by recreating and downloading the image repeatedly, but the design cycle could be reduced significantly by simply copying the new sequence to the Nintendo 64 while the application is still running.

For these applications, a host side, as well as a game side API is provided. The game side interfaces are, as always defined by including */usr/include/ultra64.h* and linking with */usr/lib/libultra[_d].a*. The host side interfaces are declared in */usr/include/ultrahost.h* and defined in */usr/lib/ultrahost.a*.

1157389

11573189

PART

Ultra 64 Operating System

115731 89

11573189

Chapter 6

Operating System Overview

1157189

Overview

The Nintendo 64 system runs under a small, real-time, preemptive kernel. It is supplied as a set of run-time library functions, so that only those portions that are actually used are included in the game's run-time image. In the remainder of this document, it is referred to as the operating system, although it is so minimal that it has not been given an official name.

The kernel can be considered as being layered into core functionality and higher-level system services, as illustrated in Figure 6-1.

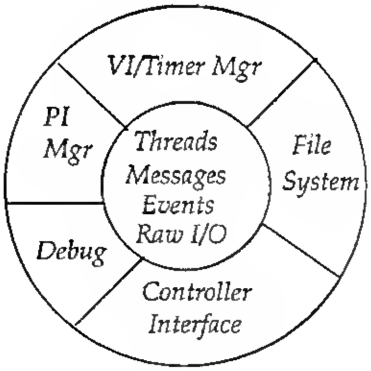

VI/Timer Mgr
PI Mgr
Threads
Messages
Events
Raw I/O
File System
Debug
Controller Interface

Figure 6-1 Nintendo 64 System Kernel

Threads, *messages*, *events*, and *raw I/O* compose the kernel of the Nintendo 64 operating system. Upon this base are built some additional services that facilitate access to the raw hardware.

In this introductory section, a brief overview of these services will be provided.

1153189

Threads

All code that runs under the operating system runs in the same address space.That is, the game runs as one process. While it is possible to structure a game application as one monolithic program, it is usually advantageous to subdivide it into smaller, more manageable subprograms called threads. With its own stack, each thread usually performs one function, often repetitively. This subdivision leads to simplicity for each thread; thus, it is easier to "get it right" and to minimize interference between threads. The threads section describes these threads, how they are scheduled, and how various operations may be performed on them.

Threads may be created, destroyed, stopped, or blocked (the latter by waiting on a message). Threads normally run until they require some resource or event to continue, at which point they yield the CPU to another thread. Each thread has an assigned priority level, used to determine which thread gets the CPU at any given time. In response to an external event, a thread may be forced to yield control of the CPU. The operating system preserves the state of the thread properly for restarting at a later time. Thus, the system can properly be described as preemptive. Threads may even be preempted during system calls when it is safe to do so.

However, there is no concept of a swap clock or "round-robin" scheduling as is found in UNIX and other time-sharing systems. Thus, two or more threads that run at the same priority level do not alternate in use of the CPU. The thread that "has" the CPU runs until it yields or is preempted by a higher priority thread in response to an exception.

Messages

Since the operating system is message-based, messages are among the most important of the resources available to the user. Unlike many popular real-time kernels, no semaphores or event flags are provided. All synchronization is provided via sending and receiving messages. This has deliberately been made very efficient, and the lack of other synchronization primitives should not be a problem. In fact, there are advantages to using only this mechanism. The operating system code itself is smaller and less intrusive on game space than it would be if it had to provide multiple facilities for thread synchronization. Also, since it is often the case that information must be transferred when threads synchronize, we get more usage out of a single operation.

Of course, messages are also useful in simply transferring information from one thread to another. In this operating system, they are also used to transfer information when a system event occurs.

Events

The operating system manages interrupts and exceptions on behalf of the game system in a relatively unobtrusive way. Some interrupts must be handled by the system code itself. Others require further decoding to determine which event has actually occurred when the CPU is interrupted.

The exception handler built into the operating system performs the decoding of interrupts and other exceptions and maps them to system events. If the system event is one that may be handled by the game itself, then a message is sent to an associated event mailbox and the game application is notified. In this way, the game designer can provide an interrupt handler to deal with the exception as required by the game requirements.

Memory Management

In this operating system, the responsibility of memory management is left up to the game. That is, the operating system provides no heap or dynamic memory allocation mechanism for the game. Since the game can access the entire memory map, it has total control on how memory is partitioned and used. The operating system simply runs in the kernel mode (kseg0) with cache and direct mapping enabled. In this mode, the virtual address 0x80000000 is mapped directly to physical address 0x0. Translation Lookaside Buffer (TLB) is not used by the operating system to provide

1153489

virtual memory support. However, low-level routines are available for game developers to program the TLBs directly. Furthermore, a region library is provided to simplify the task of allocating and de-allocating fixed-size memory buffers.

Game developers should also be aware of the importance of invalidating and flushing caches before transferring data between either cartridge ROM or RCP and main memory. The operating system provides useful functions to invalidate both instruction and data caches and to write back data cache.

Input and Output

The Nintendo 64 system spends a good deal of its time performing I/O operations. The operating system provides an optimized I/O interface layer that directly communicates with the hardware. Some of these interfaces include:

- VI—the video interface. The interface routines communicate with a video manager system thread, called the VI/Timer manager. This thread receives all vertical retrace interrupts and programs the video hardware. In addition, it also receives all counter interrupt messages and implements timer services.
- PI—the peripheral interface. The PI also has an associated I/O manager thread, the PI manager. It manages access to the ROM cartridge so that two threads do not attempt to DMA from ROM to RAM at the same time.
- AI—the audio interface. This interface programs the audio hardware to output the desired sample rate and manages access to the audio data buffer.
- DP—This is the RDP interface. It is mostly of interest because it has an associated system event when a DP operation is complete.
- Cont—the controller interface. This interface resets, detects, obtains status, queries and reads data from the game controllers.

11573189

1153789

Timers

The operating system provides convenient functions to start and stop both countdown and interval timers. These timers are expressed in CPU count register cycles, which depend on the video clock. That is, a counter tick in a PAL system occurs more frequently than the one in a NTSC system. Developers can also set and get real time counter value.

Controller Pack File System

The Nintendo 64 controller supports an add-on RAM pack that can store either 32 KB or 64 KB of data. The operating system implements a simple file system on this pack where developers can find, create, delete, read and write files.

Debugging Support

In addition to the support for the high-level GameShop debugger *gvd(1P)*, the operating system also provides additional useful facilities for debugging. Developers can use convenient routines to log messages to pre-allocated buffer for delay transfer to the host Indy. Since this logging utility has low performance impact, it may be well suited for debugging real-time problems or running performance analysis. Developers can also use the printf-like utility *osSyncPrintf(3P)* to display text formatted messages on the host Indy.

Boot Procedure

When using the Nintendo 64 development system, the developer needs to run the game loader *gload(1P)* program to download his prepared ROM image into the cartridge memory on the development board. After the memory image is loaded, gload can optionally read back the memory and verifies the contents. Then, it generates a reset signal to the development board, causing the R4300 to jump to the reset vector where it starts executing the boot code from the PIF rom.

Some of the important tasks performed by the boot code include:

1. Initialize the R4300 CP0 registers
2. Initialize the RCP (such as halt RSP, reset PI, blank video, stop audio)
3. Initialize RDRAM and CPU caches
4. Load 1 MB of game from ROM to RDRAM at physical address 0x00000400
5. Clear RCP status
6. Jump to game code
7. Execute game preamble code (which is similar to crt0.o and is linked to game during makerom process)
    - clear BSS for boot segment (as defined in the spec file)
    - set up boot segment stack pointer,
    - jump to boot entry routine
8. Boot entry routine should call *osInitialize(3P)*

11573189

1151589

Chapter 7

Operating System Functionality

Overview

Threads, messages, and events work together to form the core of the Nintendo 64 operating system. Nintendo 64 applications run under a small, multithreaded operating system. Simply put, this means that the R4300 CPU switches between several independent components called threads. Each thread consists of a sequence of instructions, a stack, and (possibly) static data that is used only by the thread. Subdividing an application into threads has several advantages. You can effectively isolate each part of the application to avoid interference. You can divide your application into small, easily-debugged modules. Since each thread can be written independently to perform exactly one function, complexity is reduced.

Messages are a mechanism by which threads communicate with one another. While this could be done using shared global variables, such an approach is often unsafe. One thread must know when it is safe to read data that is being written by another. Message passing makes communication between threads an atomic operation; a message is either available or not available, and the associated data arrives at the receiving thread at one time.

A second, perhaps more important function of messages is to provide synchronization between threads. Often a thread reaches a point in its execution where it cannot continue until another thread has completed some task. In this case, the running thread has no useful work to do, so it should yield the processor until the task is completed. You use messages to provide the mechanism for the thread to wait until that time.

Often a thread needs to wait for an exception such as an interrupt. Exceptions are trapped by the operating system and turned into events. Threads may register to receive notification of system events by requesting that the operating system send them a message whenever a system event occurs.

System Threads, Application Threads, and the Idle Thread

There are several types of threads in a typical application. There is a distinction (using priority) between system threads, application threads, and the idle thread.

The PI manager, described in the IO section, is typical of system threads. It acts as a resource manager, allowing multiple user threads to share a critical resource safely—in this case, the cartridge ROM.

The idle thread, which has the lowest priority (a priority of 0) of any thread in the system, runs only when all other threads are blocked awaiting some event. Note that the idle thread is required; the system will not run without it. The game application itself is composed of user threads. User threads are defined as those threads having priorities between 1 and 127.

Thread Data Structure

Each thread is associated with a data structure of type *OSThread* declared by the user. The address of this structure is the only identifier used in thread system calls. Since the thread data structure is essentially part of the application itself, you should take care not to overwrite it inadvertently. The structure contains the thread's context (mostly, this consists of its register contents) when the thread is not running. Each thread has a priority used in scheduling, and an identifier used only by the debugger. These are also maintained in the thread data structure.

Thread State

A thread is always in one of four states. The state of the thread is maintained in its thread data structure for use by the operating system. A good

11578089

understanding of thread state is helpful in designing your application, since it leads to a better understanding of how the operating system will behave.

- Running. Only one thread in the system is in running state at a time. This is the thread that is currently executing on the CPU.
- Runnable. A thread in runnable state is ready to run, but it is not running because some other thread has higher priority. It will gain control of the CPU once it becomes the highest-priority runnable thread.
- Stopped. A stopped thread will not be scheduled for execution. Newly created threads are in this state. Threads are frequently stopped by the debugger, and an application may stop a thread at any time. Stopped threads become runnable via an *osStartThread* system call.
- Waiting. Waiting threads are not runnable because they are waiting for some event to occur. A thread that is blocked on a message queue is in waiting state. Arrival of a message returns a waiting thread to runnable or running state.

Scheduling and Preemption

Once the OS is running, the highest-priority runnable thread in the system always has control of the CPU. When a thread gains control of the CPU, it continues to run until it requires some resource or event to continue. It then relinquishes control of the CPU and the next highest priority thread gets to run. Typically, this happens as a result of the running thread calling the function to receive a message. If no message is present in the message queue, the running thread will block until a message arrives. Note that the thread is no longer runnable when it is blocked on a message queue, so it no longer fits the criterion of being the highest-priority runnable thread.

More frequently, the running thread loses control of the CPU through preemption. In response to an exception (for example, an interrupt), a higher priority thread becomes runnable. Since that thread should now be the running thread, the state of the interrupted thread will be saved in its thread data structure, the state of the newly-runnable thread will be loaded to the CPU, and the new thread will resume execution at the point where it last ran. The preempted thread is still runnable; it just doesn't have the highest priority. When it once again becomes the highest priority thread, it will run again from the point where the interrupt occurred.

11513189

Note that the running thread does not need to be at a sequence point (for example, a system call) to lose control of the CPU. Thus, this fits the classical description of a preemptive system.

Multiple threads within an application frequently need to synchronize their execution. For example, thread A cannot continue until thread B has performed some operation. The message-passing functions provide the needed synchronization mechanism, and are described in the chapter on messages.

Thread Functions

There are eight functions associated with threads. Please refer to the reference (man) pages for specifics about the arguments, return values, and behavior of these functions.

- osCreateThread

  This function is called once per thread to notify the system that a thread is to be created. Creating a thread initializes its thread data structure with the starting program counter, initial stack pointer, and other information. Once the thread data structure has been initialized, the thread can be run.

- osDestroyThread

  This function removes a thread from the system. Once called, the thread cannot be run any more.

- osYieldThread

  This function notifies the operating system that the running thread wishes to yield the CPU to any other thread with higher or equal priority. If all other runnable threads have lower priority, the running thread will continue. (In practice, it is not possible for a runnable thread to have higher priority than the running thread.)

- osStartThread

  This function call makes a thread runnable. If the specified thread is of higher priority than the running thread, the running thread will yield the CPU. If not, the running thread will continue and the started thread will wait until it becomes the highest priority thread in the system.

- osStopThread

  This function call changes the state of a thread to stopped, after which the thread will not be able to run until restarted. If the thread was waiting on a message queue, it will be removed from that queue.

- osGetThreadId

  This function returns the ID of a thread assigned when the thread was created. It is used only by the debugger.

- osSetThreadPri

  This function changes the priority of a thread. If the running thread is no longer the highest-priority runnable thread in the system as a result of this change, it will yield the CPU to the new highest-priority thread.

- osGetThreadPri

  This function returns the running thread's priority level.

Exceptions and Interrupts

The R4300 CPU used in the Nintendo64 processes a number of exception types. Most share a common vector, where the operating system receives them, reads the CAUSE register, and determines which of the 16 legal causes occurred. With the exception of the Interrupt cause (which may be either internal or external), all exceptions are internally generated within the CPU. For example, an attempt to fetch a word from an odd address will generate an address error exception.

The operating system has exception handlers for Coprocessor Unusable, Breakpoint, and Interrupt exceptions. All other exceptions are considered to be faults and are passed to the fault handler. The fault handler stops the faulted thread, sends a message to any thread (i.e., rmon) registered for the OS_EVENT_FAULT event, and dispatches the next runnable thread from the system run queue. If the debugger is present, a message is sent from the target to the host and the debugger can show you exactly where the fault occurred. Breakpoint exceptions are also handled in this way. The debugger will stop all user threads in the event of a breakpoint or a fault.

1153189

When an interrupt occurs, the CAUSE register is examined to see which interrupt caused the exception. The R4300 supports eight interrupts described below.

Table 7-1

Name	Cause	Description
Software 1	CAUSE_SW1	Software generated interrupt 1
Software 2	CAUSE_SW2	Software generated interrupt 2
RCP	CAUSE_IP3	RCP interrupt asserted
Cartridge	CAUSE_IP4	A peripheral has generated an interrupt
Pre-nmi	CAUSE_IP5	User has pushed reset button on console
RDB Read	CAUSE_IP6	Indy has read the value in the RDB port.
RDB Write	CAUSE_IP7	Indy has written a value to the RDB port.
Counter	CAUSE_IP8	Internal counter has reached its terminal count

If the RCP interrupts the R4300, then an RCP register is read to see which of the RCP interrupts is being asserted. Thus, processing RCP interrupts is a two stage process - first the cause of the CPU interrupt is determined, then the cause of the RCP interrupt is isolated.

Normally, the Nintendo 64 game threads run with all interrupts enabled. It is possible to change the interrupt masks of the R4300 and RCP via a system call. Clearly, this must be used with great caution, as disabling a critical interrupt can cause the system to lock up or prevent real time response.

Events

Once the cause of the interrupt (or other exception) has been determined, it is mapped to one of 14 events defined for the Nintendo 64 system. Table 7-1 shows the events, why they occur, and who normally registers to receive a message when each event occurs.

Table 7-2 Events Defined for the Nintendo 64 System

Event Name	**Event Description**	**Owner**
SW1	System software interrupt 1 asserted	
SW2	System software interrupt 2 asserted	
CART	Peripherial has generated an interrupt	OS
COUNTER	Internal counter reached terminal count	VI/Timer manager
SP	RCP SP interrupt; Task Done/Task Yield	Game
SI	RCP SI interrupt; controller input available	Game
AI	RCP AI interrupt; audio buffer swap	Game
VI	RCP VI interrupt; vertical retrace	VI/Timer manager
PI	RCP PI interrupt; ROM to RAM DMA done	PI manager
DP	RCP DP interrupt; RDP processing done	Game
PRENMI	An NMI has been requested and will occur in 0.5 seconds	Game
CPU_BREAK	R4300 has hit a breakpoint	Rmon
SP_BREAK	RCP SP interrupt; RCP has hit a breakpoint	Rmon
FAULT	R4300 has faulted	Rmon
THREAD_STATUS	Thread created or destroyed	Rmon

11573169

Event and Interrupt Functions

- osSetEventMesg

  This function call specifies a message queue and message to be sent in response to a system event.
- osGetIntMask

  This function returns the current interrupt mask (including both the R4300 and RCP masks).
- osSetIntMask

  This function specifies a new interrupt mask (including both the R4300 and RCP masks).

Non-Maskable Interrupts and PRENMI

When the console RESET switch is pushed, the hardware generates a HW2 interrupt to the R4300 CPU. The interrupt is serviced by the OS event handler which sends a message of type OS_EVENT_PRENMI to the message queue associated with that event.

The HW2 interrupt will be followed in 0.5 seconds by a non-maskable interrupt (NMI) to the R4300 CPU (unless the RESET switch is pushed and held for more than 0.5 seconds, in which case the NMI will occur when the switch is released).

After the NMI occurs, the hardware is reinitialized, and:

- The first Meg of the game in ROM is copied into the first megabyte of RAM after the boot address
- The BSS for the boot segment is cleared
- The boot procedure is called.

Note: There are some minor differences between power on reset and NMI reset. After power on reset, the caches are invalidated. After NMI reset, the caches are flushed and then invalidated. Also, the power on reset configures the RAM, while NMI reset leaves the RAMs alone.

11531189

After NMI reset, the contents of memory, except for the 1 Meg that is copied in, are the same as before the NMI occured. The global variable, osResetType, is set to 0 on a power up reset and to 1 on a NMI.

If your game does not use the scheduler (see Chapter 24, "Scheduling Audio and Graphics"), it should set up to respond to the OS_EVENT_PRENMI event by associating a message queue with the event early in the game code. This is accomplished as follows:

```
osSetEventMesg(OS_EVENT_PRENMI, <some_message_queue> )
```

If your game does use the scheduler, it needs only to test for a message of type PRE_NMI_MSG on its client message queue. The scheduler performs the event initialization, and forwards the OS_EVENT_PRENMI message to the client message queue as soon as it is received.

Exactly how a game should behave when it receives OS_EVENT_PRENMI includes Nintendo policies on game consistency (such as fading the screen to black or ramping the audio volume down), but from a technical standpoint, when the game receives the OS_EVENT_PRENMI message it should do the following:

- Stop issuing graphics tasks to prevent the RDP from being stopped in a non-restartable state.
- Stop issuing audio tasks to prevent audio "pops"
- Stop issuing ROM (PI) DMAs

To test this, you can generate an NMI on development board by running the following program *on the Indy*. This is equivalent to pushing the RESET switch on the Nintendo 64 machine.

```
/*
 * Program to simulate pressing and releasing the RESET
 * switch on the Ultra 64.
 *
 * Copy this code to resetu64.c and type "make resetu64"
 *
 */
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/u64gio.h>
```

```
#include <PR/R4300.h>

#define GIOBUS_BASE     0x1f400000
#define GIOBUS_SIZE     0x200000          /* 2 MB */

main()
{
    int mmemFd;
    unsigned char *mapbase;
    struct u64_board *pBoard;

    if ((mmemFd = open(“/dev/mmem”, 2)) < 0) {
        perror(“open of /dev/mmem failed”);
        return(1);
    }

    if ((mapbase = (unsigned char *)mmap(0, GIOBUS_SIZE,
                    PROT_READ|PROT_WRITE, (MAP_PRIVATE),
                    mmemFd, PHYS_TO_K1(GIOBUS_BASE))) ==
                   (unsigned char *)-1) {
        perror(“mmap”);
        return(1);
    }

    pBoard = (struct u64_board *)(mapbase);
    pBoard->reset_control = _U64_RESET_CONTROL_NMI;
    sginap(10);
    pBoard->reset_control = 0;
}
```

Internal OS Functions

Some of the internal OS functions are briefly described below. Broken into three groups, these functions are mentioned here with the purpose to reduce potential duplicate effort from developers. Most of these functions are simple routines to access various R4300 registers, Translation-Lookaside Buffer (TLB) information, and internal active thread queue. Please refer to the reference (man) pages for specifics about the arguments, return values, and behavior of these functions.

The first group provide functions to access various common R4300 registers:

- __osGetCause, __osSetCause

11573189

These functions returns and specifies the content of the R4300 Cause register, respectively.

- __osGetCompare, __osSetCompare

  These functions returns and specifies the content of the R4300 Compare register, respectively.

- __osGetConfig, __osSetConfig

  These functions returns and specifies the content of the R4300 Configuration register, respectively.

- __osGetSR, __osSetSR

  These functions returns and specifies the content of the R4300 Status register, respectively.

- __osGetFpcCsr, __osSetFpcCsr

  These functions returns and specifies the content of the R4300 floating-point Control/Status register, respectively.

The second group provide functions to access TLB information:

- __osGetTLBASID

  This function returns the TLB Application Space ID in the R4300 EntryHi register.

- __osGetTLBPageMask

  For a specified TLB entry, this function returns the content of the R4300 PageMask register.

- __osGetTLBHi

  For a specified TLB entry, this function returns the content of the R4300 EntryHi register.

- __osGetTLBLo0

  For a specified TLB entry, this function returns the content of the R4300 EntryLo0 register.

- __osGetTLBLo1

  For a specified TLB entry, this function returns the content of the R4300 EntryLo1 register.

1157389

The third group provide functions to access internal active thread queue to find faulted thread(s):

- __osGetCurrFaultedThread

  This function returns the most recent faulted thread.

- __osGetNextFaultedThread

  This function returns the next faulted thread from the internal active thread queue.

1157318

Chapter 8

Input/Output Functionality

115318 9

Overview

The Input/Output (I/O) subsystem exists on most operating systems for three main reasons:

- to hide device-specific details in device drivers through which the operating system transfers data and control
- to provide a fair and safe access scheme to the devices, since most of them are shared resources
- to provide a consistent, uniform, and flexible interface to all devices, allowing programs to reference devices by name and perform high-level operations without knowing the device configuration.

Usually, the I/O software is structured in layers:

9. device-independent system interface
10. device drivers
11. interrupt handlers

The interrupt handler is mainly responsible for waking up a device driver after an I/O operation completes. The device driver performs device-specific operations, such as setting up registers for DMA and checking device status. The device-independent system interface provides a uniform interface to user-level software and common I/O functions (that is, protection, blocking, buffering) that can be performed across different devices.

For the RCP, there are two modes of I/O operations:

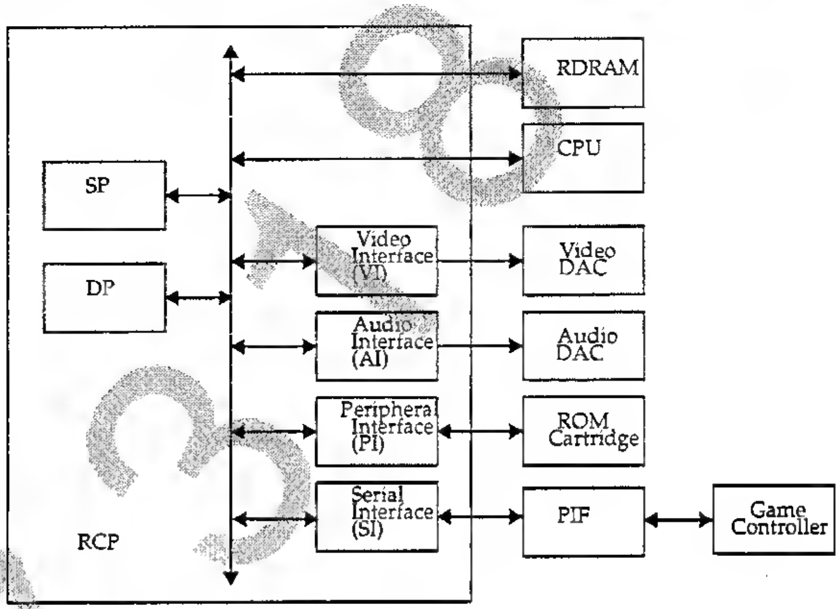
- DMA provides a minimum of 64-bit transfer between the RDRAM and any of the devices
- IO provides a 32-bit transfer between the CPU and any of the devices

The RCP consists of the following major devices and interfaces (see Figure 8-1):

- Reality Signal Processor (RSP). This internal processor supports both DMA and IO operations between RDRAM and I/Dmem addresses.
- Reality Display Processor (RDP). This internal processor supports only DMA from either RDRAM or Dmem addresses to its internal buffer.
- Video Interface (VI). This write-only interface connects to the video DAC. It supports only DMA from RDRAM to a specific video buffer address and allows you to change video modes and configurations.
- Audio Interface (AI). This write-only interface connects to the audio DAC. It supports only DMA from RDRAM to a specific audio buffer address and allows you to set the audio frequency.
- Peripheral Interface (PI). This read-write interface connects to the ROM cartridge and other mass storage devices. It supports DMA as well as IO Read/Write to ROM addresses.
- Serial Interface (SI). This read-write module interfaces to the PIF, which connects to the game controller and modem devices. It supports DMA as well as IO Read/Write to PIF RAM addresses.

115789

Figure 8-1 Logical View of RCP Internal Major Devices and Interface Modules


RDRAM
CPU
SP
DP
Video Interface (VI)
Video DAC
Audio Interface (AI)
Audio DAC
Peripheral Interface (PI)
ROM Cartridge
Serial Interface (SI)
PIF
Game Controller
RCP

Design Approach

Since Nintendo 64 operates in a real-time environment, its I/O subsystem is one of the most time-critical areas. Furthermore, the customized Nintendo 64 environment contains a well-known set of device interfaces that remains unchanged for some time to come. Therefore, its I/O subsystem is mainly designed for optimal throughput and response, and not for portability and generality. This design approach coincides with the main Nintendo 64 design philosophy, which has always been (and still is) to follow the minimal approach.

The Nintendo 64 I/O subsystem contains these components:

- a device-dependent system interface
- a device manager for shared devices

- a system exception handler

These components represent a much trimmed-down version of the typical I/O layers. All overhead associated with device-independent interfaces (that is, naming and buffering) has been removed; protection is implemented only on shared devices. Low-level (raw) I/O interface is also available, allowing you to customize device interfaces based upon your specific needs. The result is a very lightweight and optimized interface that allows you to access (in most cases) the devices directly.

Each of these components is described further in the sections below. However, first it is important to discuss some properties (such as synchrony and mutual exclusion) that the Nintendo 64 I/O subsystem should exhibit.

Synchronous I/O vs. Asynchronous I/O

Synchronous I/O and asynchronous are two fundamental methods of servicing I/O requests. In synchronous systems, the calling process is blocked after issuing an I/O request, thus allowing I/O to overlap with the execution of other processes. In asynchronous systems, the process is allowed to continue execution after initiating an I/O operation. Most systems implement the synchronous I/O method since it is easier to use and generally preferred by high-level language programmers.

However, in the Nintendo 64 environment, asynchronous I/O is the preferred choice, mainly because of the asynchronous nature of the real-time game environment. For example, a game might want to start paging in the next scene data in the background while working on the graphics task list. Therefore, asynchronous I/O has the potential to enhance the throughput on a thread basis. Furthermore, synchronous I/O can be easily implemented on top of the asynchronous facility by having the calling process blocks on a message queue immediately after initiating the I/O operation.

Therefore, all interrupt-based DMA operations are asynchronous operations and all asynchronous notification is handled via the message queue facility.

Mutual Exclusion

On most systems, some devices such as disks and printers are shared resources. The I/O subsystem must ensure that only one process can use a device at any one time, thus excluding other requesting processes and forcing them to wait.

In the Nintendo 64 environment, each device can process only one I/O transaction at any given time. For example, if there is a DMA transfer in progress between ROM and RDRAM, you cannot issue an I/O read from a different ROM location. If such a read is issued, the current DMA transaction will probably fail. Therefore, protection (or mutual exclusion) should be provided for devices that support both DMA operation and I/O read/write.

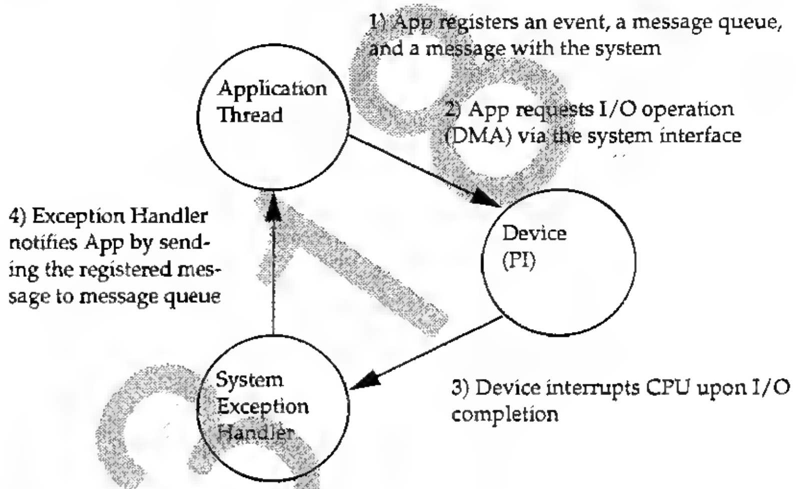
In this system, mutual exclusion is not implemented as a general scheme for all devices, but rather as a specific scheme for each identified shared device.

I/O Components

The Nintendo 64 I/O software subsystem consists of the following major components: system exception handler, device manager for shared devices, and device-dependent system interface. Figure 8-2 shows the interaction between some of these components to service an I/O request. This interaction assumes that the device is not shared, and therefore, requires no mutual exclusion.

115389

Figure 8-2 Interactions Between I/O Components Servicing Simple I/O Request


1) App registers an event, a message queue, and a message with the system
Application Thread
2) App requests I/O operation (DMA) via the system interface
Device (PI)
4) Exception Handler notifies App by sending the registered message to message queue
System Exception Handler
3) Device interrupts CPU upon I/O completion

System Exception Handler

The Nintendo 64 system contains a system-wide exception handler that traps all exceptions and interrupts. This handler is simply an optimized event notifier. That is, upon receiving an event (either a supported exception or interrupt), the handler searches the event table for an associated message queue and message, sends the message to the queue, and simply returns. The handler does not perform any device-specific operations. The `osSetEventMesg` system call is provided to register a message queue and a message with a specified event.

Device Manager

Depending on the user application, a device in the Nintendo 64 environment may be shared between two or more threads. Furthermore, if you want to utilize both DMA and IO operations on a device, you must ensure that these two operations cannot overlap. For each device that requires protection, you can use the concept of a device manager to implement mutual exclusion.

1157379

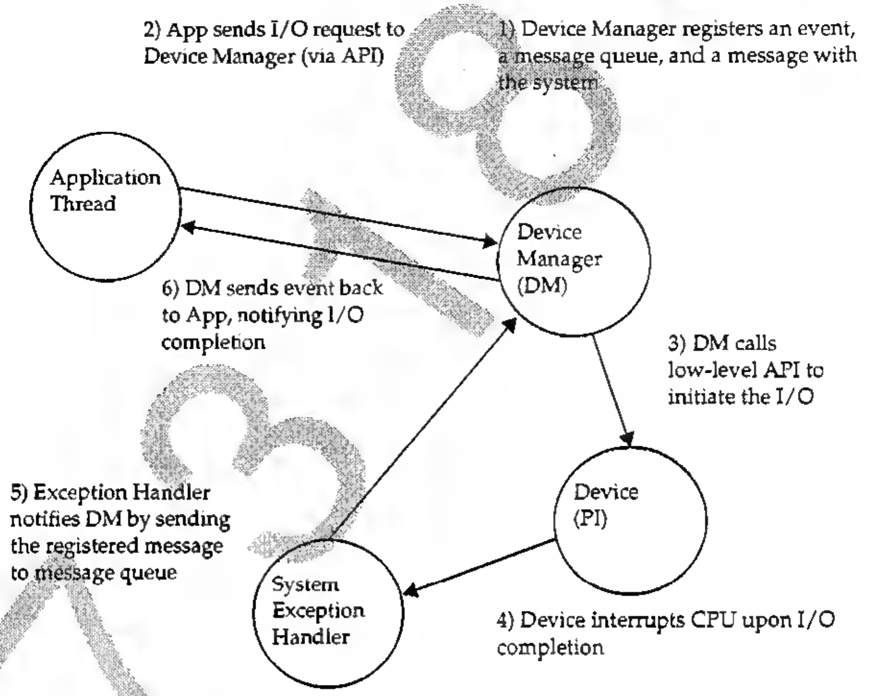
The Device Manager (DM) is simply a thread that runs at a high priority. The main purpose of this manager is to process all DMA requests to and from a device (that is, ROM devices), thus guaranteeing safe and orderly usage of the device. Upon start-up, the manager registers an event, its event message queue, and a message with the system. The manager is then blocked listening on its input command queue for request messages. The manager simply reads from the front of the queue and processes one request of a time.

After calling the corresponding low-level device routine to initiate the I/O operation, the manager then blocks on listening on the input event queue, waiting for the event sent from the exception handler, signaling I/O completion. Once awakened, the manager then notifies the calling thread (I/O requestor) by simply sending the request message to a pre-registered message queue. The manager, then, returns to listen on the input command queue for new requests.

The reason for alternating the listening between these two queues (command and event queues) is that there can be only one outstanding I/O transaction at any given time. Figure 8-3 summarizes the interaction between various I/O components to service an I/O request on a shared device.

1157889

Figure 8-3 Interaction Between I/O Components and a Shared Device


2) App sends I/O request to Device Manager (via API)
1) Device Manager registers an event, a message queue, and a message with the system
Application Thread
Device Manager (DM)
6) DM sends event back to App, notifying I/O completion
3) DM calls low-level API to initiate the I/O
5) Exception Handler notifies DM by sending the registered message to message queue
Device (PI)
System Exception Handler
4) Device interrupts CPU upon I/O completion

Device-Dependent System Interface

The device-dependent system interface is actually composed of two layers of function calls: a high-level abstraction layer and a low-level, raw I/O layer. In addition to providing mutual exclusion on devices that support both DMA and IO operations, the high-level layer also uses the lower layer to initiate raw I/O operation. The reason for exposing the raw I/O layer is to allow you to construct your own custom I/O software interface. Furthermore, if the user application requires no protection for accessing devices, using the low-level layer directly is the optimal way to request I/O operation.

1157389

11573189

In the following sections, the functions are partitioned and described under each device/interface separately. For high-level operation, each function name starts with `os<DeviceName>` for easy identification. For low-level operation, the function name starts with `os<DeviceName>Raw`. Please refer to the appropriate reference (man) pages for specifics about the arguments, return values, and behavior of these functions.

Signal Processor (SP) Functions

- osSpTaskStart

  This function loads a task and starts it running.

- osSpTaskYield

  This function asks a task running on the SP to yield.

- osSpTaskYielded

  This function checks to see if a recently completed task has yielded.

Display Processor (DP) Functions

- osDpGetStatus

  This function returns the value of the DP status register. The include file rcp.h contains bit patterns that can be used to interpret the device status.

- osDpSetStatus

  This function allows you to set various features in the DP command register. Refer to the include file rcp.h for bit patterns and their usage.

- osDpSetNextBuffer

  This function sets up the proper registers to initiate a DMA transfer from RDRAM address to the DP command buffer.

Video Interface (VI) Functions

- osCreateViManager

  This function creates and starts the VI manager (VIM) system thread.

- osViGetStatus

This function returns the value of the video interface status register. The include file rcp.h contains bit patterns that can be used to interpret the device status.

- osViGetCurrentLine

  This function returns the current half line.

- osViGetCurrentMode

  This function returns the current VI mode type.

- osViGetCurrentFramebuffer

  This function returns the currently displaying frame buffer.

- osViGetNextFramebuffer

  This function returns the next frame buffer to be displayed.

- osViGetCurrentField

  This function returns the current field (either 0 or 1) being access by VI manager.

- osViSetMode

  This function sets the VI mode to one of the possible 28 modes. The new mode takes effect at the next vertical retrace interrupt.

- osViSetEvent

  This function registers a message queue with the VI manager to receive the notification of a vertical retrace interrupt.

- osViSet[X/Y]Scale

  These two functions allow you to change the horizontal scale-up factor (x-scale) and vertical scale-up factor (y-scale), respectively.

- osViSetSpecialFeatures

  This function enables/disables various special mode bits in the control register.

- osViSwapBuffer

  This function registers the frame buffer with the VI manager to be displayed at the next vertical retrace interrupt.

115731 89

Audio Interface (AI) Functions

- osAiGetStatus

  This function simply returns the value of the audio interface status register. The include file rcp.h contains bit patterns that can be used to interpret the device status.

- osAiGetLength

  This function simply returns the number of bytes remained in the audio interface DMA length register.

- osAiSetFrequency

  This function configures the audio interface to support the requested frequency (in Hz). It calculates necessary values to program internal divisors and returns the closest frequency that the divisors can generate.

- osAiSetNextBuffer

  This function programs the next DMA transfer based on the input length and starting buffer address.

Peripheral Interface (PI) Functions

- osCreatePiManager

  This function creates and starts the PI manager (PIM) system thread.

- osPiGetStatus

  This function simply returns the value of the hardware status register. The include file rcp.h contains bit patterns that can be used to interpret the peripheral status (that is, DMA busy and IO busy).

- osPiRawStartDma

  This low-level function sets up the proper registers to initiate a DMA transfer between ROM and RDRAM.

- osPiRaw[Read/Write]Io

  These two low-level functions perform an IO (32-bit) read/write from/to ROM address space, respectively.

- osPi[Read/Write]Io

These two functions perform IO (32-bit) read/write from/to ROM address space, respectively. Since they provide mutual exclusion for accessing the PI device, these routines are both blocked I/O calls.

- osPiStartDma

  This function generates an asynchronous I/O request to the PI manager to initiate a DMA transfer between RDRAM and ROM address space. Upon I/O completion, PI manager notifies the requestor by returning the I/O request message to the message queue specified by the requestor.

Controller Functions

- osContInit

  This function initializes all the game controllers and returns a bit pattern to indicate which game controllers are connected.

- osContReset

  This function resets all game controllers and returns their joysticks to neutral position.

- osContStartQuery

  This function issues a query command to all game controllers to obtain their status and type.

- osContGetQuery

  This function returns the game controllers' status and type.

- osContStartReadData

  This function issues a read data command to all game controllers to obtain their input settings.

- osContGetReadData

  This function returns the game controllers' joystick data and button settings.

1153189

Chapter 9

Basic Memory Management

Introduction

This chapter

- describes the hardware and software features of the Nintendo 64 platform that relate to memory management, and
- discusses how an application may use them for efficient, correct memory utilization and access.

The software interface of the Nintendo 64 platform allows you to take advantage of the hardware capabilities of the machine, which include high flexibility and high performance. However, with this flexibility comes a corresponding decrease in ease of programming, which this chapter addresses.

Hardware Overview

Recall that the primary processing elements of the machine are the MIPS R4300 CPU and the Reality CoProcessor (RCP). The CPU executes application code directly from the DRAM, transparently caching instruction and data references in on-chip caches. The code itself makes references to CPU virtual addresses, which are translated by on-chip hardware to physical memory addresses.

1153189

The RCP is primarily composed of two elements: the Signal Processor (SP) and the Display Processor (DP). The SP is a microcoded engine that processes task lists for audio and graphics. The DP is, for the most part, driven by the SP. The RCP can be treated as a single processor for the purposes of memory management.

Finally, a number of DMA engines also access DRAM directly: the DP, as well as the Audio Interface (AI), Serial Interface (SI), and Parallel Interface (PI).

At the hardware level, all of these agents make references to physical DRAM addresses. These physical addresses are derived in very different ways, however.

CPU Addressing

CPU virtual address translation takes place in either of two ways: either via direct mapping or through the translation lookaside buffer (TLB). When running in kernel mode (as applications do on the Nintendo 64 platform) the address ranges have the behavior described in Table 9-1.

Table 9-1 32 Bit Kernel Mode Addressing

Beginning	**Ending**	**Name**	**Behavior**
0x00000000	0x7fffffff	KUSEG	TLB mapped
0x80000000	0x9fffffff	KSEG0	Direct mapped, cached
0xa0000000	0xbfffffff	KSEG1	Direct mapped, uncached
0xc0000000	0xdfffffff	KSSEG	TLB mapped
0xe0000000	0xffffffff	KSEG3	TLB mapped

The KSEG0 address space is expected to be the most popular, if not only, address space used. In this address space, the physical memory locations corresponding to be KSEG0 address can be determined by stripping off the upper three bits of the virtual address. For example, virtual address 0x80000000 corresponds to physical address 0x0000000, and so on.

SP Addressing

The SP microcode makes address references also, but these references are only to the local memory (IMEM and DMEM) on the chip. With the current software architecture, the application does not program the SP directly, and need not concern itself with IMEM and DMEM accesses.

DRAM references, however, concern the application, because large data structures stored in DRAM are passed by reference. These include matrices, vertex lists, textures, and the display lists themselves. As for the CPU, the addresses given to be SP for these data objects are also virtual addresses, but the mapping from virtual to physical address is significantly different. The SP microcode maintains 16 locations in DMEM that act as segment base registers. An "SP virtual" address is presented to the SP microcode in the form of a <segment number, segment offset> pair encoded into a 32-bit word. To compute a physical DRAM address, the microcode adds the contents of the corresponding segment base register to the given offset.

DMA Engine Addressing

As indicated above, the Nintendo 64 includes DMA engines that access DRAM directly. Since these DMA operations are initiated by the CPU, the DRAM addresses passed to the interface routines are CPU virtual addresses. These routines perform the mapping from virtual to physical addresses and give the resulting physical DRAM address to be appropriate hardware registers.

Makerom and Memory Management

In addition to its more obvious role of creating the application ROM image, *makerom* (1P) is a powerful tool for both memory and symbol table management. Segments to *makerom* mean more than SP addressable memory regions. To *makerom*, a segment is any contiguous, coherent region of bytes in memory or on the ROM.

The ROM specification file given to *makerom* provides virtual or segment addresses to segments. A segment consisting of MIPS 4300 code or data to run on the CPU can be given a virtual address with an `address` statement. A segment consisting of static display list data is given a segment address by specifying the segment number with a `number` statement.

Briefly, *makerom* does the following:

- scans the input specification file for syntax errors;
- sizes the segments, creating absolute symbols for segment addresses and ROM locations;
- performs final relocations of relocatables that comprise the segment, using a link editor that can link an arbitrary number of segments to different addresses;
- extracts the text and initialized data portions for each segment from the resulting fully linked binary, and packs these portions of the segment onto the ROM image.

Mixing CPU and SP Addresses

It is permissible to link segments given a CPU virtual address with those given a SP segment address. It may appear counter-intuitive and error-prone to link relocatables of entirely incompatible address spaces. As it turns out, the benefits outweigh the potential risks, because it allows the application code to address SP display list data symbolically.

For example, suppose a segment is composed of the following display list data:

```
static Vp vp = {
        SCREEN_WD*2, SCREEN_HT*2, G_MAXZ/2, 0,/* scale */
        SCREEN_WD*2, SCREEN_HT*2, G_MAXZ/2, 0,/* translate */
};

Gfx rspinit_dl[] = {
    gsSPViewport(&vp),
    gsSPClearGeometryMode(0xffffffff),
    gsSPSetGeometryMode(G_SHADE | G_SHADING_SMOOTH),
    gsSPEndDisplayList(),
};
```

The beginning of the display list `rspinit_dl` is embedded somewhere in the segment. Rather than computing its offset into the segment, the display list is simply provided symbolically:

```
   gSPDisplayList(glistp++, rspinit_dl);
```

The compiler and linker do the work of computing the address of `rspinit_dl` within the segment. Thus, if the relative location of the display list `rspinit_dl` changes, the code will still remain valid (and more readable). Note that the CPU does not reference any of the data in this display list; the CPU just passes a reference to the display list data to the SP.

A more complicated example involves using the mixed symbol table to work with memory regions created by the CPU and read by the SP. In this case, a single SP segment refers to two different underlying DRAM regions. This technique can be useful when static display lists need to refer to dynamic data that is double buffered. The actual DRAM location currently being pointed to is swapped by setting the appropriate SP segment register.

The actual memory for the dynamic data can be declared and created within a KSEG0 code segment as follows:

```
typedef struct {
                 Mtx projection;
       Mtx modeling;
       Gfx glist[2048];
   } Dynamic_t;

   Dynamic_t dynamicBuffer[2];
   Dynamic_t *dynamicPointer = &dynamicBuffer[0];
```

The segment contents can then be modified by the CPU directly:

```
   guOrtho(&dynamicp->projection,
             -SCREEN_WD/2.0, SCREEN_WD/2.0,
             SCREEN_HT/2.0, SCREEN_HT/2.0, 1, 10, 1.0);
   guRotate(&dynamicp->modeling, theta, 0.0, 0.0, 1.0);
```

The SP view of the dynamic segment is created by creating a relocatable with the following parallel definition and assigned to, for example, segment register 4 in the ROM specification file:

1157389

```
Dynamic_t rspdynamic;
```

Since the relocatable contains only uninitialized data (bss), no actual bits on the ROM are used. But more importantly, the symbol *rspdynamic* is made available to other objects. Its value is the segment address of the dynamic segment.

The SP segment register 4 is then mapped to the actual memory for the dynamic segment with the following command:

```
gSegment(glistp++, 4, osVirtualToPhysical(dynamicp);
```

Then the SP addresses of the dynamic structure can be used, even from static display lists, to build display lists that reference components of the dynamic section:

```
    gsSPMatrix(&dynamic.projection,
          G_MTX_PROJECTION|G_MTX_LOAD|G_MTX_NOPUSH);

    gsSPMatrix(&dynamic.modeling,
          G_MTX_MODELVIEW|G_MTX_LOAD|G_MTX_NOPUSH);
```

As with the previous example, using the compiler and linker to generate addresses allows the data structures to be modified, reordered, and so on, without changes to unaffected areas of the application.

Flushing the CPU Data Cache

The MIPS R4300 CPU transparently caches data accesses on a onboard data cache. Ordinarily this cache is of no concern to the application, but when an external agent such as the SP or DMA engine is involved, the application must be aware of the caching implications.

The data cache implements a "write back" replacement policy which means that data stores are held in the cache until the entire cache line is written back, usually due to a cache miss thatrequires the same cache line. The cache is not coherent with respect to physical memory and thus cache lines must be explicitly written back to memory prior to their use by another processor such as the SP.

115189

Using the above example, the dynamic data can be written with a single procedure call as follows. It is expected that this will be done prior to the task list being executed by the SP.

```
    osWritebackDCache(dynamicp, sizeof(Dynamic_t));
```

Clearing uninitialized data (Bss) section

Prior to loading a segment into memory, the application must invalidate the corresponding cache lines. The *makerom(1P)* makes appropriate symbols available to the application that can be used to construct the arguments to the *osInvalDCache(3P)* routines. Then the actual DMA from ROM to DRAM may be performed, as well as the clearing of the uninitialized data (bss) section of the segment. It is important that the clearing be performed before the Bss section can be used. Again, *makerom(1P)* generated symbols may be used for the *bzero()* call. Here is some sample code that illustrates the process:

```
extern char _newSegmentRomStart[], _newSegmentRomEnd[];
extern char _newSegmentStart[];
extern char _newSegmentDataStart[], _newSegmentDataEnd[];
extern char _newSegmentBssStart[], _newSegmentBssEnd[];

   osInvalDCache(_newSegmentDataStart,
       _newSegmentDataEnd-_plainSegmentDataStart);
   osPiStartDma(&dmaIOMessageBuf, OS_MESG_PRI_NORMAL,OS_READ,
       (u32)_newSegmentRomStart, _newSegmentStart,
       (u32)_newSegmentRomEnd - (u32)_newSegmentRomStart,
       &dmaMessageQ);

   bzero(_newSegmentBssStart,
         _newSegmentBssEnd-_newSegmentBssStart);

(void)osRecvMesg(&dmaMessageQ, NULL, OS_MESG_BLOCK);
```

Physical Memory Allocation

The Nintendo 64 hardware contains four megabytes of "nine bit" DRAMS. The normally hidden ninth bit is used for antialiasing and z-buffering hardware. It is recommended that the framebuffer and z-buffer reside on *different* megabyte banks to take advantage of caching in the DRAM circuitry

By default, the boot location resides at directed mapped address 0x80000400. (or physical address 0x400). The first 1024 (0x400) bytes of physical memory are reserved for exception vectors and configuration parameters. This boot location can be changed by simply inserting an address statement in the boot segment of the *makerom (1P)* specification file. For example, the following code specifies the boot location to be at 0x80200000, which is the beginning of the third megabyte of memory.

```
beginseg
        name "code"
        flags BOOT OBJECT
        entry boot
        address 0x80200000
        stack bootStack + STACKSIZE
        include "codesegment.o"
        include "$(ROOT)/usr/lib/PR/rspboot.o"
        include "$(ROOT)/usr/lib/PR/gspFast3D.o"
        include "$(ROOT)/usr/lib/PR/gspFast3D.dram.o"
        include "$(ROOT)/usr/lib/PR/aspMain.o"
endseg
```

The boot process of the Nintendo 64 will copy one megabyte of data beginning with the boot segment specified in the specification file to the boot location.

1157189

Chapter 10

Advanced Memory Management

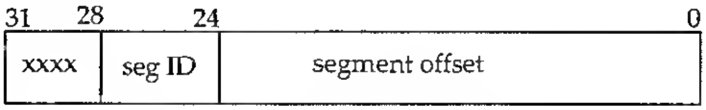
11573189

Introduction

This chapter explores techniques and features that are not required in the simplest of applications. It contains useful information and tricks that may be used in certain situations, but it is not expected that all applications will use all the techniques described here.

Mixing CPU and SP Data

In the previous chapter it was implied that CPU and SP data should be in separate segments as they are addressed differently. This is not mandatory, however, as the addressing can be easily reconciled. Suppose the application defines a display list and includes it in a segment given a CPU addressable KSEG0 address. The physical address of this display list can be easily determined with the *OS_K0_TO_PHYSICAL(3P)* macro or the *osVirtualToPhysical(3P)* routine. The resulting physical address corresponds to an SP address with segment number if 0, and a segment offset equal to the physical address. This is because the encoding of the SP segment address is as follows:



If the application creates a mapping using segment 0 to a beginning physical address of 0x0, the SP can correctly access objects in DRAM when given a physical address.

This simplifies the situation somewhat, but the SP microcode takes it a step further: Since the upper four bits of a segment address are not used, they are ignored. Thus an implicit mapping is done from a KSEG0 address to a physical address, and no explicit conversion need be done by the application.

To summarize, as long as an SP segment table mapping is done from segment number 0 to offset 0, CPU KSEG0 addresses can be interpreted correctly by the SP.

Using Overlays

The total application code size and data will probably be greater than what is actively being used at any point in time. To conserve DRAM, applications may choose to only have active code and data resident. To facilitate this, the application can be partitioned into a number of segments, where some segments share the same memory region during different phases of execution. Here is an excerpt from a specification file that contains a kernel code segment that can call routines in either of two overlay segments, *texture* and *plain*:

115319

```
beginseg
        name "kernel"
        flags BOOT OBJECT
        entry boot
        stack bootStack + STACKSIZE
        include "kernel.o"
        include "$(ROOT)/usr/lib/PR/rspboot.o"
        include "$(ROOT)/usr/lib/PR/gspFast3D.o"
endseg

beginseg
        name "plain"
        flags OBJECT
        after "kernel"
        include "plain.o"
endseg

beginseg
        name "texture"
        flags OBJECT
        after "kernel"
        include "texture.o"
endseg

beginwave
    name "overlay"
    include "kernel"
    include "plain"
    include "texture"
endwave
```

Note the use of the *after* keyword to place both of the overlay segments at the same address.

Prior to loading a segment into memory, the application must invalidate the corresponding instruction and data cache lines. The *makerom(1P)* makes appropriate symbols available to the application that can be used to construct the arguments to the *osInvalICache(3P)* and *osInvalDCache(3P)* routines. Then the actual DMA from ROM to DRAM may be performed, as well as the clearing of the uninitialized data (bss) section of the segment. Again, *makerom(1P)* generated symbols may be used for the *bzero()* call. After

11573189

the segment is loaded, any procedure in the segment may be called or any data in the segment referenced. Here is some sample code that illustrates the entire process:

```
extern char _plainSegmentRomStart[], _plainSegmentRomEnd[];
extern char _plainSegmentStart[];
extern char _plainSegmentTextStart[], _plainSegmentTextEnd[];
extern char _plainSegmentDataStart[], _plainSegmentDataEnd[];
extern char _plainSegmentBssStart[], _plainSegmentBssEnd[];

    osInvalICache(_plainSegmentTextStart,
        _plainSegmentTextEnd-_plainSegmentTextStart);
    osInvalDCache(_plainSegmentDataStart,
        _plainSegmentDataEnd-_plainSegmentDataStart);
    osPiStartDma(&dmaIOMessageBuf, OS_MESG_PRI_NORMAL,OS_READ,
        (u32)_plainSegmentRomStart, _plainSegmentStart,
        (u32)_plainSegmentRomEnd - (u32)_plainSegmentRomStart,
        &dmaMessageQ);

    bzero(_plainSegmentBssStart,
          _plainSegmentBssEnd-_plainSegmentBssStart);
(void)osRecvMesg(&dmaMessageQ, NULL, OS_MESG_BLOCK);
```

Using Multiple Waves

The previous example linked both overlays into a single, fully relocated binary. This binary is used for two purposes. First, the text and data sections are extracted from this binary and packed on the ROM. Second, this binary can be given to the Nintendo 64 debugger, *gvd(1P)*. Although the specification file above will create an operationally correct ROM image, the binary will confuse the debugger. This is because multiple symbols will map to the same address, and *gvd* may err when it tries to find the correct source line for a given program counter value, for example.

This problem can be circumvented by creating multiple binaries, or *waves*, each with a distinct symbol table. The following specification file excerpt illustrates this:

```
beginwave
        name "plain_wave"
        include "kernel"
        include "plain"
```

```
endwave

beginwave
        name "texture_wave"
        include "kernel"
        include "texture"
endwave
```

Using this technique, procedure and variable names from the *plain* segment are kept distinct from those of the *texture* segment. The "Switch Executable" menu entry from the *gvd* "Admin" menu can be used to select the symbol to use while debugging.

There is one significant caveat when using multiple waves. The contents of each segment must be identical in each of the waves the segment is included in. For example, the *kernel* segment above is included in both *plain_wave* and *texture_wave*, so its relocated image must be identical in both. The usual consequence of this rule is that the segment procedure entry point in both of the overlay segments must be at the same location. This requirement can be easily met by ensuring that the segment procedure is always the first procedure of the first relocatable that comprises the overlay segment. Then the calling segment code can always jump to the beginning address of the overlay segment(s) and execute valid code there.

Using the Region Allocation Routines

Previous examples were primarily concerned with static memory allocation; many applications may find it necessary to do some form of dynamic allocation. For situations where the allocation is always done in fixed size chunks, a family of region allocation routines are provided. These routines will carve up a larger buffer into fixed some memory regions that are managed by the library. The routines of interest are:

- osCreateRegion

  This function initializes an allocation arena given a memory address, size, and alignment.

- osMalloc

This function allocates and returns the address to a single fixed sized and properly aligned buffer from a given region. This function will fail and return NULL is there is no available free buffer in the region.

- osFree

  This routine returns a previously allocated buffer to the given region pool.
- osGetRegionBufCount

  This function returns the total number of buffers in the region.
- osGetRegionBufSize

  This function returns the actual buffer size, after having been possibly padded to the given alignment.

The following code sample creates a region, allocates a buffer, and then frees it.

```
void *region;
char regionMemory[REGION_SIZE];
u64 *buffer;

region = osCreateRegion(regionMemory,
                        sizeof(regionMemory),
                        BUFFER_SIZE, OS_RG_ALIGN_16B);
buffer = osMalloc(region);

/* do some work that uses 'buffer' */

osFree(region, buffer);
```

Incidentally, if the fixed size regions are intended to hold entire segments, the *maxsize* keyword of the makerom specification file may be of interest. See *makerom(1P)* for details.

Managing the Translation Lookaside Buffer

Although most applications will find the direct mapped KSEG0 address space of the CPU sufficient, it is possible to use the mapped address space by setting appropriate Translation Lookaside Buffer (TLB) entries.

11573189

1157389

Perhaps the biggest restriction with using the TLB is that individual entries operate only on relatively large, aligned memory regions (pages). Nevertheless, it may be helpful for memory protection or relocation of CPU addresses. In addition, TLBs can be used as yet another method to reconcile SP segment addresses with CPU addresses, since SP addresses fall within the range of the mapped CPU address space.

The translation lookaside buffer (TLB) of the R4300 has 32 entries, each of which maps two physical pages. The TLB is fully associative, which means each entry is essentially independent—the index number implies nothing about the mapping and any entry can hold any mapping. A number of page sizes are supported: 4 KB, 16 KB, 64 KB, 256 KB, 1MB, and 16MB. Each TLB entry may map a different page size. The following routines are used to manage the TLB:

- osMapTLB

  This function sets the contents of a single TLB entry to the given virtual address, even and odd physical address, page size, and address space identifier.

- osUnmapTLB

  This function invalidates both the odd and even physical page mappings of a given TLB entry.

- osUnmapTLBALL

  This function invalidates all mappings in the TLB. This should be done by the application prior to using the TLB.

- osSetTLBASID

  This function sets the current address space identifier register.

Using the TLB requires some care. The following paragraphs describe some problem areas.

- Two TLB entries cannot map the same virtual address space. If this occurs, accesses to the address will cause a TLB refill exception. Any overlapping mapping creates this condition, even when a mapping with a smaller page size is a subset of another mapping with a larger page size:

  ```
  osMapTLB(0, OS_PM_16K,(void *)0x0,0xa0000,-1,-1);
  osMapTLB(1, OS_PM_4K, (void *)0x2000, 0xb000, -1, -1);
  ```

Another case involves different TLB entries, each of which map different pages of an odd/even pair. The following mappings, which individually map an even and an odd physical page, will create an overlap condition:

```
osMapTLB(0, OS_PM_4K, (void *)0x2000, 0xa000,-1, -1);
osMapTLB(1, OS_PM_4K, (void *)0x2000,-1, 0xb000, -1);
```

Instead, the application should set a single entry with both mappings:

```
osMapTLB(1, OS_PM_4K, (void *)0x2000, 0xa000, 0xb000, -1);
```

- The mapped addresses must be aligned to the page size. This applies to both the virtual and physical pages mapped.

  This implies that if one intends to map SP segment addresses via the TLB, the SP segment must be loaded at a page-aligned address.

- Multiple mappings of a cached address must be of the same "color." CPU caches are physically tagged, but virtually indexed, which introduces a situation in which more than one cache line references the same physical memory locations. Avoid the problem by using the same virtual address consistently for a particular physical address.

  If you cannot use the same virtual address, the mappings should all be the same color, where the "color" is defined as bits [14..6] of the instruction address (for instruction fetches) or bits [15..5] of the data address (for data accesses).

Finally, no support is provided for handling and recovering from TLB misses. A TLB miss is an unrecoverable fault to the Nintendo 64 system.

More information about these topics can be found in the MIPS R4300 documentation.

PART

Ultra 64 Graphics

115731889

11573189

Chapter 11

Graphics Microcode

Graphics are rendered in Nintendo64 games by creating a graphics display list, and passing this display list to the RSP. In order for the RSP to process this display list, the application, using system calls, loads graphics microcode. This section discusses the different microcode object files available to applications.

There are six basic versions of the graphics microcode, and each basic version has up to three subtypes. The basic versions are know as, gspFast3D, gspF2DNoN, gspLine3D, gspTurbo3D, gspSuper3D, gspSprite2D. Each basic version has a different set of graphics rendering features. Each subtype has the same set of graphics features, but varies according to how the RSP passes commands to the RDP. The three subtypes are regular, .dram and .fifo. The object files for the microcode are labeled, <basicType>.o, <basicType>.dram.o, and <basicType>.fifo.o.

11540189

Microcode Functionality

gspFast3D

gspFast3D microcode is the most full-featured of the microcode objects. It is also the microcode used in the majority of the demo applications. gspFast3D supports 3D triangles, 3D clipping, z-buffering, near and far clipping, lighting, mip-mapped textures, perspective textures, fog, and matrix stack operations. It does not support the GBI command, gSPLine3D.

gspF3DNoN

The gspF3DNoN microcode is similar to the gspFast3D microcode, except it does not handle near plane clipping in the same manor. When using the gspFast3D microcode, objects between the eye and the near plane are clipped. When using the gspF3DNoN microcode, objects between the eye and the near plane are not clipped. However, the area between the eye and the near clipping plane does not implement zbuffering. This means that objects that fall into this area must be drawn in order from far to near.

gspLine3D

gspLine3d microcode features many of the features of gspFast3D, except instead of drawing triangles, it draws 3D lines. This is useful for producing wireframe effects. If a gSP1Triangle command is encountered it will draw the three edges of the triangle, but not the center portion of the triangle.

gspTurbo3D

gspTurbo3D microcode is a reduced-feature, reduced-precision, microcode that delivers significantly faster performance. The features not supported by gspTurbo3D are: Clipping, lighting, perspective-corrected textures, and matrix stack operations. The quality of the anti-aliasing also suffers, due to the lack of precision used by gspTurbo3D. This loss of precision can also manifest itself as various visual artifacts, depending on the content. gspTurbo3D uses a different format for the display list.

gspSprite2D

gspSprite2D microcode is optimized for drawing 2D sprite images. Sprites are implemented as textured screen rectangles. gspSprite2D does not support 3D lines 3D triangles, vertices operations, matrix operations, lighting, or fog. All of the DP commands such as blender modes, and color combiner modes are supported. Zbuffering can be used to arrange the order of the sprites from front to back

gspSuper3D

gspSuper3D is a reduced precision microcode that supports the same display list format as gspFast3D. This reduced precision will increase performance, but can cause visual artifacts. Although gspSuper3D uses the same display lists as gspFast3D, gspSuper3D does not support perspective corrected textures.

1157369

RSP to RDP command passing

All types of RSP microcode generate commands for the RDP. The method used to pass the commands from the RSP to the RDP determines the suffix used to name the microcode object. In the "regular" method the commands are written to a buffer in dmem, which can hold up to six RDP commands. If the buffer fills, the next time the RSP tries to write a command it will stall until there is space in the buffer. Microcode versions that use this type of command passing have no special suffix, just a ".o" appended to their name.

Alternatively, the RSP can write all the commands to a larger fifo buffer in rdram. This helps to prevent the RSP from stalling when the RDP gets bound by processing large triangles. Microcode that uses this method has the ".fifo.o" suffix appended to its name.

When using the fifo version of a microcode, the application must pass a pointer to a buffer to be used as the fifo buffer, in the task output_buff field. The size of the fifo buffer is put in the output_buff_size field. In order for fifo to have a positive effect on performance the size of the buffer should be greater than 1K.

The microcode also provides another option for the RSP to write all of the RSP commands to an rdram buffer. In this case the application must start the RDP task separately with a call to `osDpSetNextBuffer()`. (This form of command passing is very useful for debugging in conjunction with the tool dlprint which can print display lists in a human readable form.) Microcode designed to use this method has the ".dram.o" suffix appended to its name.

Tasks using the .dram microcode need a pointer to a buffer in the output_buff field of the task structure, and a size in the output_buff_size. Because RSP commands usually expand when converted into RDP commands, this buffer needs to be larger than the size of the RSP display list.

Chapter 12

RSP Graphics Programming

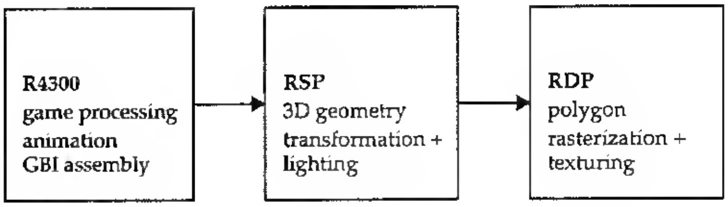
11573189

This document describes the graphics state machine of the RCP, with a particular focus on the RSP (see "RSP: Reality Signal Processor" on page 44).

The RSP is an R4000-like CPU with an 8-element vector unit, featuring a small instruction memory, IMEM (4K bytes or 1K instructions) and small data memory, DMEM (4K bytes). Software running on this processor implements a large portion of the geometry display pipeline.

In addition, the RSP provides visibility for all of the RCP functionality, through a variety of software conventions and hardware exposure. All "display lists" for the RCP graphics features must pass through the RSP. There are several important features which require the application programmer to be consciously aware of the distinctions between the RSP and the RDP (and program each of them separately), but for the most part, the RSP serves as the single interface between the application program and the graphics pipeline:

Figure 12-1 Nintendo 64 Graphics Pipeline


R4300
game processing
animation
GBI assembly
RSP
3D geometry
transformation +
lighting
RDP
polygon
rasterization +
texturing

Topics covered in this document include:

- RSP overview
- display list processing
- matrix state
- vertex state
- vertex lighting state
- texture state
- clipping and culling
- primitives
- controlling the RDP state

1157318

1157389

RSP Overview

A program which runs on the RSP is called a *task;* the application is completely responsible for scheduling and invoking tasks on the RSP.

The interface between the application and the RSP task is accomplished with a series of operating system calls, and a structure called the *task list* (or task header) which is type *OSTask* (defined in *sptask.h*). The task list contains all the information necessary to begin task execution, including pointers to the microcode to run. This structure is filled in by the application program.

A detailed description of invocation of a task on the RSP is beyond the scope of this section (see "RCP Task Management" on page 65), but the essential procedure is straightforward:

- the RSP is assumed to be halted (or the R4300 halts it).
- the R4300 DMA's the boot microcode into the RSP IMEM.
- the R4300 DMA's the 'task header' into the RSP DMEM.
- the R4300 sets the RSP PC to 0.
- the R4300 clears the RSP halt status (allowing it to run).

From this point, the boot microcode takes over, loading the task microcode (and data) specified in the task list, and jumping to the beginning of the task.

One item in the task header is a pointer to the initial data to process (in the case of a graphics task, this is a display list pointer).

Display List Format

The display list which the *gspFast3D, gspF3DNoN, or gspLine3D* microcode running on the RCP interprets is defined as a stream of 64-bit commands.

Applications written in C will usually use the interface from the file *gbi.h.*, which will be included via inclusion of *ultra64.h.* Although the construction of display lists looks like a familiar series of function calls, they are actually just bit-packing macros. These macros are described in detail in their individual man pages.

Each macro has two forms, i.e. *gSPTexture()* and *gsSPTexture()*. The difference between *'g'* and *'gs'*, is that the *'g'* form is an in-line form which requires an additional argument (pointer of the display list being constructed). The display list pointer must be of the form *"ptr++"*, in order for the macros to work properly.

The *'gs'* form is for static declarations, and generates the appropriate C structure initialization sequence.

Throughout this document, only the *'gs'* form is mentioned, however the *'g'* form also applies, and could always be substituted.

All of the display list building macros also embed an *'SP'* or a *'DP'* to describe the functional unit of the RCP which will operate on this command. This is certainly confusing, especially to application programmers familiar with higher-level graphics API's such as OpenGL. In order to achieve maximum performance, it is necessary to expose the two major units of the RCP to the application programmer. The primary reason for this is resource constraints; there is simply not enough RSP IMEM to build a display list processor that is rich enough to hide these details from the application programmer. In addition, given the dedicated application of the RCP (video games), any CPU cycles spent "gift-wrapping" the graphics API are a waste of time. The binary encoding of most of the display list commands *is* the lowest possible level: they *are* the bits that control the hardware.

Exposing the two functional units of the RCP also limits the amount of state shared between them. The major drawback of this design decision is that you must often tell the same thing to the RSP and the RDP. For example, in order to "turn on texture mapping" you must turn it on in the RSP *and* turn it on in the RDP. This may seem clumsy at first, and indeed this is a common source of display list bugs, but the parallel execution of the RSP and RDP, plus the lean display list processing machine make this trade-off worthwhile.

Segmented Memory and the RSP Memory Map

All DRAM addresses in the display list are segmented addresses. The mapping of segments and their base addresses is provided using the `gsSPSegment()` macro. It is the responsibility of the application to maintain this mapping and inform the RSP via the display list.

1153489

The RSP maintains an associative table of up to 16 segment ID's and their base addresses. Any DRAM address in the display list is 'physical-ized' using this table.

The RDP only uses physical addresses, and one of the chores of the RSP is to do the address translation necessary for the RDP.

Note: By convention, segment table entry 0 is reserved for physical addressing, and should be set to 0x0.

The RSP software can only access DMEM. All data must first be transferred into DMEM using DMA operations, which must be 64-bit aligned. Invocation of the DMA engine is handled by the RSP software, but the application programmer needs to be aware of the boundary requirements. Any data structure that is to be passed to the RSP must be aligned to a 64-bit boundary. The structures in *gbi.h* use C unions to guarantee this.

Since the DMA engine is shared between the R4300 and the RSP, the application program should also avoid unnecessary DMA activity while the RSP is running.

Interaction Between the RSP and R4300 Memory Caching

The most prevalent example of communication between the CPU and the RSP is that of the CPU creating a display list in DRAM for eventual interpretation by the RSP. The display list data is read from DRAM via a DMA mechanism. Unfortunately, DRAM locations may be "stale" with respect to newer data being held in the R4300's data cache. The R4300 cache mechanism implements a "write-back" caching policy which means individual stores to memory are not immediately written to memory. To update the memory contents with more recent cached data, the CPU must first write back cached data to the DRAM. Then, and only then, will the RSP be able to DMA the correct data for display list processing.

Conversely, the contents of memory may be more recent than cached data in some situations when the RSP modifies memory (an obvious example is updating the color frame buffer). In this case, the CPU's cache may contain stale data and the CPU should invalidate the cached data to force an access directly to DRAM and get the most recent data.

As a practical note, this second scenario only arises in advanced applications.

11573189

Display List Processing

Understanding the basics of the RSP display list processing is necessary to construct efficient, compact display lists for an application.

The display list (or command list) can be thought of as a hierarchical structure, up to 10 levels deep. A display list may contain a pointer to another display list, and so on. The RSP processes the display list using a stack, pushing and popping the current display list pointer.

For animation, it will be desirable to "double-buffer" parts of the display list; rendering one frame while the data for the next frame is updated. In this case, only the minimum amount of data need be duplicated; only the data which will change for each frame. Swapping between doubled buffers is efficiently done by changing the segment base addresses (and organizing your display list appropriately).

During computation by the RSP, all display lists and their data must remain in the same location until the RSP is finished. This sounds obvious, but is a very common bug, usually the result of incorrect usage of double-buffering techniques. In addition, if the RSP task is interrupted (see "Signal Processor (SP) Functions" on page 109), all of the data must remain in the same location when/if the task is restarted

Connecting Display Lists

Hierarchical display list connection can be made with the *gsSPDisplayList()* macro. The current display list location is pushed on the display list stack and processing begins with the new display list.

Table 12-1 gsSPDisplayList(Gfx *dl)

Parameter	**Values**
dl	pointer to the display list to attach.

1157189

Branching Display Lists

A display list branch without a push allows you to "chain" together fragments of display lists for more efficient memory utilization.

Table 12-2 gsSPBranchList(Gfx *dl)

Parameter	Values
dl	pointer to the display list to attach.

Ending Display Lists

All display lists must terminate with an "end" command.

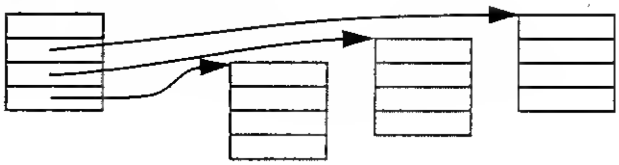
Table 12-3 gsSPEndDisplayList(void)

Parameter	Values
none	none

A Few Words about Optimal Display Lists

The display list processor running on the RSP caches display list commands in groups of about 32. This means the optimal display list size is a multiple of 32. A display list of 33 commands (or 65, etc.) would require the display list cache to be refilled during processing, possibly causing a wait state (depending on the DMA engine activity). Obviously not all display lists can keep the list processor running 100% optimally, but it is something to keep in mind when tuning your application.

Another form of display lists which cause less than optimal processing are display lists that look like this:



Since the display list engine is stack-based, a display list that has lots of unnecessary indirect pointers will cause lots of unnecessary pushes and pops, which do have a cost.

Constructs like this are unavoidable sometimes, like when sharing geometries among objects, but if you have a choice try not to group indirect display list pointers together.

115731899

Matrix State

The "geometry engine" in the RSP implements a fixed-point matrix engine with the following matrix state:

A 10-deep modeling matrix stack. New matrices can be loaded onto the stack, multiplied with the top of the stack, popped off of the stack, etc. This matrix stack is primarily used for manipulating objects within the world coordinate system (often combinations of rotations, translations, and sometimes scales).

A 1-deep projection and viewing matrix "stack". New matrices can be loaded onto the stack, multiplied with the top of the stack, but cannot be pushed or popped. This matrix "stack" is primarily used for the projection matrix and the viewing matrix. The projection matrix (often created with the guPerspective or the guOrtho functions) is loaded onto the stack, and then the viewing matrix (often created with the guLookAt function) is multiplied on top of it.

A "perspective normalization" factor. This is used to improve precision of the fixed-point perspective computation.

When a group of vertices is loaded, they are first transformed by the matrix MP (the current top of the modeling stack multiplied by the projection matrix). All vertex transformations are done only when they are loaded; sending a new matrix down later will not change any points already in the points buffer.

The modeling matrix stack resides in DRAM. It is the application's responsibility to allocate enough memory for this stack and provide a pointer to this stack area in the task list.

The format of a matrix is a bit unusual. It is optimized for the RSP's vector unit (used during the multiplies and transformations.) This format groups all of the integer parts of the elements, followed by all of the fractional parts of the elements. This unusual format is not exposed to the user, unless he/she chooses not to use the matrix utilities in the libraries.

Insert a Matrix

Inserts a new matrix into the display list.

Table 12-4 gsSPMatrix(Mtx *m, unsigned int p)

Parameter	Values
m	pointer to the new matrix.
p	G_MTX_MODELVIEW or G_MTX_PROJECTION, G_MTX_MUL or G_MTX_LOAD, G_MTX_PUSH or G_MTX_NOPUSH

Pop a Matrix

This command pops the matrix stack.

Table 12-5 gsSPPopMatrix(unsigned int n)

Parameter	Values
n	unused

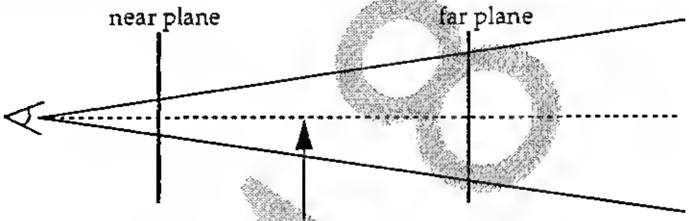
Perspective Normalization

This scale value is used to scale the transformed *w* coordinate down, prior to dividing out *w* to compute the screen coordinates (which are similarly scaled). The effect of this is to maximize the precision of this divide.

11573189

The library function *guPerspective()* returns one approximation for this scale value, which is a good estimate for most cases:

Figure 12-2 Perspective Normalization Calculation



$$\frac{1}{s} = \frac{(near + far)}{2}$$

so $s = \frac{2}{(near + far)}$ (represented as an unsigned 16-bit fraction)

This approximation normalizes w=1.0 halfway between the near and far planes.

Table 12-6 gsSPPerspNormalize(unsigned short int s)

Parameter	Values
s	16-bit unsigned fractional perspective normalization scale.

Note on Coordinate Systems and Big Numbers

The RSP is a fixed point machine, so keeping coordinate systems within a certain range is important. If numbers in the final coordinate system (or intermediate coordinate systems) are too big, then the geometry of objects can be distorted, textures can shift erratically, and clipping can fail to work correctly. In order to avoid these problems keep the following notes in mind:

1) No coordinate componant (x, y, z, or w) should ever be greater than 32767.0 or less than -32767.0

2) The difference between any 2 vertices of a triangle should not have any componants greater than 32767.0

1151389

3) The sum of the difference of w's of any 2 vertices plus the sum of the differneces of any of the x, y, or z componants should be less than 32767.0. In other words for any 2 vertices in a triangle, v1=(x1,y1,z1,w1), and v2=(x2,y2,z2,w2) , these should all be true:

```
    abs(x1-x2) + abs(w1-w2) < 32767.0
    abs(y1-y2) + abs(w1-w2) < 32767.0
    abs(z1-z2) + abs(w1-w2) < 32767.0
```

One way to check this is to take the largest vertices that you have and run them throught the largest matrices you are likely to have, then check to make sure that these conditions are met.

A reccommended way of avoiding trouble is to never allow any componant to get larger than 16383.0 or smaller than -16383.0. To ensure this find:

M = the largest componant (x, y, or z) of the largest model in your database.

S = The largest scale (ie number in the upper 3 rows of the matrix) in the matrix made up of the concatenation of the largest modeling matrix, the largest LookAt matrix, and the largest Perspective matrix you will use.

T = the largest translation (ie number in the 4th row of the matrix) in the matrix made up of the concatenation of the largest modeling matrix, the largest LookAt matrix, and the largest Perspective matrix you will use.

Now $M * S + T < 16383.0$ should be true. If you experience textures wobbling or shifting over a surface, clipping not working correctly, or geometry behaving erratically, this is a good place to check.

A Few Words About Matrix Precision

The RSP uses fixed-point 32-bit multiplies during matrix operations. Since the product of two 32-bit numbers is a 64-bit number, only the middle 32 bits of the answer is retained. Overflow of intermediate terms is possible, especially in large coordinate systems or unusual projection matrices.

In order to avoid fixed-point precision problems, in some cases it may be desirable to compute the matrix in floating point on the R4300 and just load it.

Matrix multiplies are very fast on the RSP, but they are not free. If possible, reduce matrix operations by pre-multiplying the matrices at modeling time or compile time.

11573189

Vertex State

The RSP state includes a vertex buffer, holding up to 16 vertices. This buffer can be loaded with any number of consecutive vertices, beginning at any location.

Table 12-7 gsSPVertex(Vtx *v, unsigned int n, unsigned int v0)

Parameter	Values
v	pointer to a list of vertices.
n	number of vertices.
v0	vertex buffer location to load vertices into.

At the time the vertices are loaded, they are transformed by the current matrix state and possibly shaded by the current lighting state.

Vertices are not re-transformed again, if the matrix state changes, the old (previously-transformed) vertices are not affected. This feature can be exploited to construct data that is knit together between two groups of points with different transformations (such as an elbow joint of a character).

Since the vertex processing is heavily vectorized and pipelined, it is important that each load loads as many vertices as possible.

Since the vertex loading is a relatively slow operation, it is also important that any triangles that share vertices be rendered using the same vertex state, rather than re-loading these same vertices later.

See the "Note on Coordinate Systems and Big Numbers" on page 146 for info on keeping your coordinates from becoming too big.

115389

Texture State

The following command sets the RSP texture state:

Table 12-8 gsSPTexture(int s, int t, int levels, int tile, int on)

Parameter	Values
s	*s*-coordinate texture scale (16-bit unsigned fraction)
t	*t*-coordinate texture scale (16-bit unsigned fraction)
levels	(maximum number of mip-map levels) - 1
tile	which tile in the TMEM
on	G_ON or G_OFF

As explained previously, a vertex's *s* and *t* coordinates are texel-space coordinates in a S10.5 format. The texture coordinate usually ranges from 0 to (texel_size - 1), possibly larger to implement "wrapped" textures. The maximum number of times that a texture may be wrapped is limited by the number of integer bits in this coordinate.

Since the *s* and *t* coordinate texture scale parameters are only fractional numbers, they cannot represent values >= 1.0. For non-scaled textures, applications typically use a vertex texture coordinate format of S9.6, and a scale value of 0.5 (0x8000 in 16-bit unsigned format).

The *levels* parameter tells the pipeline the maximum number of mipmap levels to use, if mip-mapping is enabled.

The *tile* parameter tells the pipeline which of the 8 possible tiles in the RCP texture memory to use when texturing the following primitives

The *on* parameter turns texturing on or off in the RSP. If texturing is turned off in the RSP, textured primitives will not be generated, regardless of the RDP state.

Likewise, setting the RSP state is necessary, but not sufficient to generate textured primitives. The RDP state must also be set in the appropriate manner, see "TX: Texture Engine" on page 186.

Texturing is sensitive to large numbers and overflows. Refer to the Note on Coordinate Systems and Big Numbers in the Matrix State section for notes on how to avoid texturing problems such as textures shifting across surfaces, textures tearing, and edges between polygons becoming visible in the texture.

115731809

1153789

Clipping and Culling

3D clipping is automatically enabled all the time. There are two modes which can be adjusted for performance and appearance: ClipRatio and NearClipping. See also "Scissoring" on page 184.

3D clipping is expensive and should be avoided. Methods employed by the host application which can reduce the amount of geometry that gets clipped are a good idea. Crude visibility determination algorithms, geometric level-of-detail, and careful scene construction can help improve clipping performance dramatically.

The clipping algorithm is sensitive to large numbers and overflows. Refer to the **Note on Coordinate Systems and Big Numbers** in the **Matrix State** section for notes on how to avoid clipping problems.

Clip Ratio

The Clip Ratio feature helps the application to clip less.

Generally (ie when ClipRatio is set to FRUSTRATIO_1) the RSP clips to the clipping frustrum which is defined by the projection and viewing matrices (often created using guPerspective and guLookAt respectively). This is the area which is mapped by the gSPViewport command and usually corresponds to the entire frame buffer. Objects outside this area are scissored by the RDP, so clipping them is not neccessary. The ClipRatio command can set the area which is clipped between 1 and 6 times the size of the viewing frustrum. Polygons which are completely on the screen are drawn without clipping. Polygons which are partially onscreen but completely within the enlarged frustrum are drawn without clipping (the extra portions are scissored away). Polygons which are entirely offscreen are trivially rejected (whether they are inside or outsid the frustrum). The only polygons which are clipped are the large polygons which stretch all the way from onscreen to outside the enlarged clipping boundary. There is some overhead for drawing sections of polygons which are then scissored away, but it is much smaller than the time to draw actual onscreen pixels and is usually faster than clipping. Different values of ClipRatio can be tried to obtain the best performance. High values of ClipRatio are suspected to be associated with "texture shuffle" bugs, so if you see the texture shuffling you could try lower values of ClipRatio.

To set the ClipRatio so that the clipping frustrum is 3x the size of the screen:

```
gsSPClipRatio(FRUSTRATIO_3),
```

You can use values of FRUSTRATIO_1, FRUSTRATIO_2, ..., FRUSTRATIO_6

Near Clipping and gspF3DNoN microcode

3D clipping causes geometry which is outside of a 3D box called the "clipping Frustrum" to be clipped away (ie not rendered). The left, right, top and bottom of this clipping frustrum box correspond to the left, right, top, and bottom of the screen. However the side facing towards the viewer and the side facing away from the viewer do not correspond to physical parts of the screen. The "far plane" is the side of the box farthest from the viewer. Objects which are farther away than this plane are not rendered. Likewise the "near plane" is the side of the box closest to the viewer. Objects which are closer to the viewer than this plane are not rendered. The near and far clipping planes can cause visual problems. Objects which get too far away will suddenly dissappear as the cross the far clipping plane. Also, objects which get too close to the viewer will suddenly dissappear as the cross the near clipping plane.

There is a solution to these problems. The near plane problem can be partially solved by using the gspF3DNoN microcode (which is an acronym for Fast 3D No Near clipping). The gspF3DNoN microcode will not clip objects between the viewer and the near clipping plane (objects which would have been clipped away by the gspFast3D microcode). However, Z buffering will not work correctly in this area. Objects between the viewer and the near plane will hide objects which are behind the near plane, but objects between the viewer and the near plane will not correcly hide other objects between the viewer and the near plane. For this reason it is important for the application to ensure that only one object at a time comes closer to the viewer than the near plane.

There is a solution to the far plane problem too. Objects which get farther away from the viewer than the far plane visually "pop" out of view, and objects approaching the viewer "pop" into view. The Fog effect can be used to make objects gradually fade into a distant fog, or slowly appear through a distant fog, instead of popping into and out of view. See the **Vertex Fog State** section for details.

115789

Back-Face Polygon Culling

The geometry engine of the RSP implements a flexible polygon culling algorithm; either the front-facing, the back-facing, neither, or both types of polygons can be culled before rasterization.

This offers the programmer the most database flexibility. Geometry can be ordered in any direction or re-used with different culling flags in order to achieve effects such as interior surfaces, 2-sided polygons, etc..

Table 12-9 gsSPSetGeometryMode(unsigned int n)

Parameter	Values
n	G_CULL_FRONT G_CULL_BACK G_CULL_BOTH

Table 12-10 gsSPClearGeometryMode(unsigned int n)

Parameter	Values
n	G_CULL_FRONT G_CULL_BACK G_CULL_BOTH

Volume Culling

The RCP can perform volume culling. The volume of an object is described to the RCP and the RCP only draws the object if the described volume is entirely or partially onscreen. If the volume is entirely offscreen then the display list is quickly skipped.

The volume of an object is described with a number of vertices surrounding the object. The vertices may be part of the object or not. They can be 4 vertices describing a pyramidal volume, 8 points describing a cube, or any other convex shape. These vertices should be sent to the RCP using a gSPVertex command just like regular vertices (note: you may want to turn lighting and fog off when these vertices are sent for better performance). Then the gsSPCullDisplayList command is sent. If the volume is entirely off the screen then the command acts like gsSPEndDisplayList and the rest of

1157389

the display list is skipped. Otherwise the command acts as a NOOP and the display list processing continues.

1157318 9

1153169

Vertex Lighting State

The RCP graphics pipeline provides a number of sophisticated real-time lighting effects, including ambient (uniform) lighting, diffuse (directional) lights, specular highlights, and automatic texture coordinate generation (fog is discussed in its own section later). To achieve these effects and perform the lighting operations, the following steps must be carried out:

1) Reference the gspFast3D microcode in the "spec" file.

2) Replace colors with normal components in the vertices of objects to be rendered.

3) Define light structures with the parameters of the directional and ambient lights and send them to the RCP.

4) Modify the state of the RCP to "turn on" lighting.

5) Define a texture map of the shape of the specular highlights to be used and describe them to the RCP.

6) Define structures with the parameters of specular highlights and send them to the RCP.

7) Render the objects.

Steps 1), 2), 3), 4), and 7) are required for diffuse and ambient lighting. All steps are required for specular lighting. These steps are described in further detail below.

RSP Microcode

Lighting requires the gspFast3D or gspF3DNoN microcode. This microcode must be referenced in the "spec" file when the rom image is created. The part of the microcode that performs the lighting calculations is not normally resident, but is brought in through an overlay when lighting calls are made. This has performance implications for rendering scenes with some objects lighted and others colored statically. Moreover, the lighting overlay overwrites the clipping microcode, so to achieve highest performance, it is best to minimize or avoid completely clipped objects in lighted scenes.

Normal Vector Normalization

To light an object, the vertices which make up the object must have normals instead of colors specified. The normal consists of 3 signed 8-bit numbers representing the x, y, and z components of the normal. Each component ranges in value from -128 to +127. The x component goes in the position of the red color of the vertex, the y into the green, and the z into the blue. Alpha remains unchanged. The normal vector must be normalized. This means that square_root(x*x + y*y + z*z)== 127. To normalize the normal (x,y,z) determine d=127/square_root(x*x + y*y + z*z). Then form XN=x*d; YN=y*d; ZN=z*d. The normalized normal vector is (XN,YN,ZN). (Note the libultra/gu square_root function is sqrtf().)

Ambient and Directional Lighting

Lighting helps achieve the effect of depth by altering the way objects appear as they change their orientation. The RSP microcode supports up to 7 directional lights and 1 ambient light in a scene. Each directional light has a direction and a color. Ambient lights have color only. Regardless of the orientation of the object and the viewer, each directional light will continue to shine in the same direction (relative to the "world") until the light direction is changed. In addition, one ambient light provides uniform illumination. Shadows are not explicitly supported.

Important note on Matrix Manipulation

It is important, when lighting, that the projection matrix and the viewing matrix (ie matrices which describe the view into the world coordinate system) be placed on the projection matrix stack(G_MTX_PROJECTION), while matrices used to describe the position and orientation of objects within the world coordinate system are placed on the modeling matrix stack (G_MTX_MODELVIEW).

Light Structure Definition

Lighting information is passed to the RSP in light structures. Since the number of diffuse lights can vary from 0 to 7, there are 8 macros used to define lights: gdSPDefLights0, gdSPDefLights1, gdSPDefLights2, ... , gdSPDefLights7. The number which is the last character in the macro signifies the number of diffuse lights in the scene. Correspondingly, the number of diffuse lights to be rendered determines which macro to use in defining the light structure. There is always one ambient light.

To define a light structure use gdSPDefLights# where # is the number of diffuse lights to be turned on. For example, for 3 lights:

```
Lights3 light_structure1 = gdSPDefLights3(
        ambient_red, ambient_green, ambient_blue,
        light1red, light1green, light1blue,
                   light1x, light1y, light1z,
        light2red, light2green, light2blue,
                   light2x, light2y, light2z,
        light3red, light3green, light3blue,
                   light3x, light3y, light3z);
```

will define a structure called light_structure1 with an ambient light and 3 directional lights. The variables with red, green, blue suffixes represent the color of the light and take on values ranging from 0 to 255. The variables with the x, y, z suffixes represent the direction of the light and take on the range from -128 to +127. The light direction does not need to be normalized. The convention is that the light direction points toward the light. This means the light direction indicates the direction **TO** the light and **NOT** the direction that the light is shining. Note the direction the light is shining is the negative of the light direction. For example if the light is coming from the upper left of the world, the direction might be x=-80, y=80, z=0. If this diffuse light is green, and the ambient light is red, this structure would be defined by:

```
Lights1 my_light = gdSPDefLights1(
        /* ambient color red */
        255, 0, 0,
        /* green light from the upper left */
        0, 255, 0,   -80, 80, 0);
```

To avoid any ambient light, make the ambient light black (0,0,0). To include only ambient light, and no diffuse directional light, use gdSPDefLights0:

```
Lights0 my_ambient_only_light = gdSPDefLights0(
        /* blue ambient light */
        0, 0, 255);
```

Note on Light Direction

The light direction does not need to be normalized. However, there are some problems that can arise from using light directions with magnitudes that are too large or too small. The Light direction is multiplied times the Modelview Matrix (actually the transpose of the model matrix). If the Modelview matrix has a scale associated with it then the light direction might overflow or underflow. If the Modelview matrix has a scale S associated with it and the magnitude of the light direction is L then you should ensure that

```
1 < L*S < 23040
```

in order to keep the light working consistantly. If L*S is too big then the normalization of the lights will overflow and you will get lights that are too bright. If L*S is too small then the nortmalization will underflow and you will get lights that are too dim. Note the number 23040 comes from the formula: (L/128)*S < sqrt(32768) because the result of the matrix multiply of L (which is a s.7 number, thus the /128) times the matrix (thus S, the scale of the matrix, which is an s15.16 matrix) must produce a number which can be squared (thus the square root) to produce a number which is s.15 (up to 32768).

Lighting State Set Up

To activate a set of lights in a display list use the macros: gsSPSetLights0, gsSPSetLights1, gsSPSetLights2, ... , gsSPSetLights7. For example, the following macros would activate the lights defined in the examples above

```
gsSPSetLights3(light_structure1), or
gsSPSetLights1(my_light), or
gsSPSetLights0(my_ambient_only_light),
```

in a static display list. (To activate the lights in a display list dynamically the corresponding gSPSetLights# macros would be used.) Once lights are activated, they will remain on until the next set of lights is activated. This implies that setting up a new structure of lights overwrites the old structure of lights in the RSP.

To turn on the lighting computation so that the lights can take effect, the lighting mode bit needs to be turned on. This is accomplished using the macro:

```
gsSPSetGeometryMode(G_LIGHTING)
```

Object Rendering

Objects are rendered by issuing geometric primitive commands (see Primitives section). The objects drawn will use lighted colors instead of vertex colors. This means any color combiner mode will use lighted colors in the combination operation in a manner exactly analogous to vertex color use in non-lighted rendering. Note that lighting is performed at Vertex processing time. Therefore it is important that lighting state be established prior to gSPVertex and gsSPVertex commands describing vertices in a lit primitive. Lighting state established between a gSPVertex command and a gSP1Triangle command will have no effect on that triangle.

NOTE ON MATERIAL PROPERTIES

Material properties are not explicitly supported. Instead material colors and light colors have been combined in the Light structure. To obtain the correct light color in a particular situation, multiply the the color of the material times the color of the light foreach light source and use the result as the lights color. Since colors range from 0 to 255, the result will have to be normalized by dividing by 255 in order to obtain a resulting light color in the 0 to 255 range. In other words, if your material color is (mr, mg, mb) and your light is (lr,lg,lb), then the light color you would use would be (mr*lr/255, mg*lg/255, mb*lb/255). For example to light a purple object (color=255,0,255) with yellow ambient light (color=255,255,0) and cyan directional light (color=0,255,255) you could use:

```
Lights1 material1_light = gdSPDefLights1(
        /* ambient color red = purple * yellow */
        255, 0, 0,
        /* blue directional light = purple * cyan */
        0, 0, 255,   -80, -80, 0);
```

If you then want to change the material color (eg to light an object of different color) you can define a 2nd Light structure with different light colors but the same directions and send it to the RCP after the first object's vertices and before the second objects vertices. For example to light a second object which is yellow (color=255,255,0) with the same yellow and cyan light as above you could use:

11539

1157389

```
Lights1 material2_light = gdSPDefLights1(
        /* ambient color yellow = yellow * yellow */
        255, 255, 0,
        /* green directional light = yellow * cyan */
        0, 255, 0,    -80, -80, 0);
```

PERFORMANCE NOTE: the gsSPSetLights# macros incur a certain overhead when they are called in order to recalculate the new position of the light. If the colors of the lights are being altered but the directions will remain the same you can use the gSPLight macro to send the new light structure after the 1st primitives vertex command and before the second primitive's. Note that the directional lights are always referred to as lights 1-N (where N is the number of directional lights in the scene) and the ambient light is always referred to as light N+1. For the example above, the entire sequence would look like:

```
gsSPSetGeometryMode(G_LIGHTING),
gsSPSetLights3(material1_light),
gsSPVertex( /* define vertices for object 1 */ );
/* render object 1 here */
gsSPLight(&material2_light.l[0], LIGHT_1),
gsSPLight(&material2_light.a, LIGHT_2),
gsSPVertex( /* define vertices for object 2 */ );
/* render object 2 here */
```

Specular Highlights

A specular highlight is the bright spot that shiny objects exhibit when the viewing direction lines up properly with a highly directional light source.It is caused by the light from the light source being directly reflected into the eye of the observer. A specular highlight appears on a shiny object wherever the normal of the object bisects the angle between the direction of the light and the direction of the eye. The gspFast3D microcode can support zero, one, or two specular highlights on an object. If there are more than 2 lights in a scene, a quite impressive specular highlight effect can still be achieved by choosing the two most important lights and rendering the highlights from them. Specular highlights use texture mapping so specular highlights cannot usually be used with texture mapped surfaces. Specular highlighting when combined with diffuse lighting (described above) can produce very realistic looking surfaces. While specular highlighting is not required to be on when diffuse lighting is on, diffuse lighting must be on when specular lighting is on. However, the specular highlights do not neccessarily have to correspond to the diffuse lights at all.

A specular highlight is basically a reflection of a light source. To render it on the RCP requires a texture map of an image of the light. The specular highlight from most lights can be represented by a round dot with an exponential or gaussian function representing the intensity distribution. If the scene contains highlights from other, oddly shaped lights such as fluorescent tubes or glowing swords, the difficulty in rendering is no greater provided a texture map of the highlight can be obtained. The center of the image of the light should be in the center of the texture map and the texture map must be a power of 2 in width and height. In general shinier objects reflect smaller, sharper highlights. A dull object might have a large white dot for a specular highlight whether it is lit by a glowing sphere or a flaming sword. A shiny metallic object would reflect the sword as a picture of the sword and the texture map used for highlighting different types of objects can portray this difference. Note that many objects, such as human skin and cloth, which reflect specular highlights to some extent, often can benefit more from a regular texture map (eg hair on the body or a pattern on the cloth. Since these materials are not shiny the texture mapping ability may be better spent on a conventional textutre map.

Specular Highlight Structure Definition

Specular lighting information is passed to the RSP in structures, analogous to the diffuse light case. The utility procedure guLookAtHilite fills in the elements of 2 structures, Hilite and LookAt, for use in highlighting. To accomplish this, the two structures must be part of the dynamic segment, declared as

```
Hilite hilite;
LookAt lookat;
```

and guLookAtHilite must be called for each object in the following manner:

```
guLookAtHilite(&throw_away_matrix, &lookat, &hilite,
               Eyex,      Eyey,     Eyez,
               Objectx,   Objecty,  Objectz,
               Upx,       Upy,      Upz,
               light1x,   light1y,  light1z,
                    light2x,   light2y, light2z,
                    tex_width, tex_height);
```

where the arguments in common with guLookAt have the same meaning. Objectx, Objecty, and Objectz are the world coordinates of the center of the object. light1x, light1y, and light1z are the direction of the light which is reflected in the 1st highlight (should be the same as the direction specified in the gdSPDefLights# macro). light2x, light2y, and light2z are the direction of the light which causes the second highlight (if you are only using one highlight these may be zero). tex_width and tex_height are the size of the texture to be used for the highlight and must be powers of 2.

The information in the LookAt structure is sent to the RSP with the LookAt macro:

```
    gsSPLookAt( &lookat ),
```

Texture Loading

The texture for the highlights must be loaded with gsDPLoadTextureBlock or similar loadblock command. For example, the following call loads a tex_width by tex_height 4-bit intensity texture:

```
    gsDPLoadTextureBlock_4b(hilight_texture, G_IM_FMT_I,
                            tex_width, tex_height, 0,
                            G_TX_WRAP | G_TX_NOMIRROR,
                            G_TX_WRAP | G_TX_NOMIRROR,
                            tex_width_power2,
                            tex_height_power2,
                            G_TX_NOLOD, G_TX_NOLOD),
```

where tex_width_power2, tex_height_power2 are the logarithms to the base 2 of the texture width and height. Note that wrapping must be turned on, and the texture sizes must be a power of 2 for proper operation. The texture loadblock macro sets a texture tile with the parameters necessary for rendering one texture, and thereby one of the specular highlights. Setting a second texture tile with the parameters for rendering a second specular highlight can be done by loading another texture, but generally the same texture can be used for both highlights. Instead, setting up a second tile if the specular highlights are sharing one texture map can be accomplished with a set tile call. The example following assumes the same 4 bit intensity texture as used for the first highlight:

1157319

```
gsDPSetTile(G_IM_FMT_I, G_IM_SIZ_4b,
                        ((tex_width/2)+7)>>3,
                          0, G_TX_RENDERTILE+1, 0,
                          G_TX_WRAP | G_TX_NOMIRROR,
                          tex_width_power2, G_TX_NOLOD,
                          G_TX_WRAP | G_TX_NOMIRROR,
                          tex_height_power2, G_TX_NOLOD),
```

Texture Coordinate Transformations

Specular highlighting utilizes the projection of the vertex normals in the x and y directions in screen space to derive the s and t indices respectively for referencing the texture. The normals must be normalized as described above. The normal projections are scaled to obtain the actual s and t values for the reference. The scaling is applied in the RSP. It maps the negative most projection of a unit normal, or -1, into zero. It maps the positive most projection, or +1, into a scale value passed in through the gsSPTexture command. Suppose the maximum texture s, t coordinates are tex_s_max and tex_t_max. The following command sets the scale, so that a normal project of +1 in the x direction in screen space will be mapped with the texel with s coordinate tex_s_max:

```
gsSPTexture((tex_s_max)<<6, (tex_t_max)<<6, 0,
                            G_TX_RENDERTILE, G_ON),
```

The left shift of argument by 6 bits is done to account for the S10.5 16-bit internal representation of the texture coordinates (see Texture State below) and a multiplication by one-half in the microcode.

Highlight Position Description

After the texture is loaded, the highlight position information must be sent to the RSP. This information is contained in the Hilite structure, and is sent to the RSP with the following macros:

```
gsDPSetHilite1Tile(G_TX_RENDERTILE,&hilite,
                    tex_width, tex_height),
gsDPSetHilite2Tile(G_TX_RENDERTILE+1,&hilite,
                    tex_width, tex_height),
```

where both highlights share the same texture.

Lighting State Set Up

Specular highlighting requires the lighting and texture generation mode bits to be turned on using the macro:

```
gsSPSetGeometryMode(G_LIGHTING | G_TEXTURE_GEN),
```

Object Rendering

As with diffuse lighting, objects are rendered by issuing geometric primitive commands (see Primitives section). For two specular highlights, the 2 cycle mode can be used, with a cycle devoted to each highlight. In addition, since each highlight can have a different color, two registers are needed to hold the colors for combining. The Primitive Color register holds the first highlight's color and the Environment register holds the second highlight's color. As an example, the following calls:

```
gsDPSetCycleType(G_CYC_2CYCLE),
gsDPSetEnvColor(0, 255, 255, 255),          /* cyan   */
gsDPSetPrimColor(0, 0, 255, 255, 0, 255),    /* yellow */
gsDPSetRenderMode(G_RM_PASS, G_RM_AA_ZB_OPA_SURF2),
gsDPSetCombineMode(G_CC_HILITERGBA, G_CC_HILITERGBA2),
```

set up rendering of a cyan and an yellow highlight in opaque z-buffered antialiased mode. Note that for most materials the highlight color is the same as the light's color, in contrast to the diffuse light case where the resultant color is often affected by the color of the object it is striking (although metallic objects like gold and brass usually have material-colored highlights).

Reflection Mapping

Reflection mapping maps a texture onto an object using the normals of the object to specify where on the object the texture will be mapped. If this texture is an image of the surroundings of the object, then this rendering will make the object appear to reflect its surroundings. This effect simulates the rendering of objects made of chrome or having a highly reflecting, mirror-like surface.

Structure Definition

As with diffuse and specular lighting, information for reflection mapping is passed to the RSP in a structure. The utility procedure guLookAtReflect fills in the elements of a LookAt structure for use in reflection mapping. To accomplish this, the structure must be part of the dynamic segment, declared as

```
LookAt lookat;
```

and guLookAtReflect must be called for each object in the following manner:

```
guLookAtReflect(&throw_away_matrix, &lookat,
                Eyex,     Eyey,     Eyez,
                Objectx,  Objecty,  Objectz,
                Upx,      Upy,      Upz      );
```

where the arguments in common with guLookAt have the same meaning. Objectx, Objecty, and Objectz are the world coordinates of the center of the object.

The LookAt structure contains information about the orientation of the object relative to the viewing direction. This information is sent to the RSP with the LookAt macro:

```
gsSPLookAt( &lookat )
```

Texture Loading

The texture for reflection mapping must be loaded with a loadblock command such as gsDPLoadTextureBlock, described in the example above. As in the specular highlighting case, wrapping must be turned on, and the texture sizes must be a power of 2 for proper operation.

Texture Coordinate Transformations

Reflection mapping utilizes the projection of the vertex normals in the x and y directions in screen space to derive the s and t indices respectively for referencing the texture. The normals must be normalized as described above. The normal projections are scaled to obtain the actual s and t values for the reference. The scaling is applied in the RSP. It maps the negative most

1153789

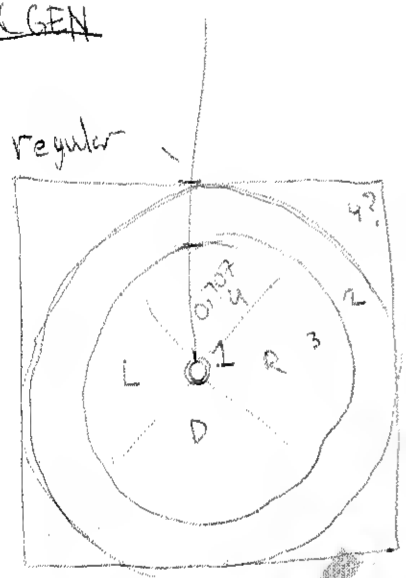
projection of a unit normal, or -1, into zero. It maps the positive most projection, or +1, into a scale value passed in through the gsSPTexture command. Suppose the maximum texture s, t coordinates are tex_s_max and tex_t_max. The following command sets the scale, so that a normal project of +1 in the x direction in screen space will be mapped with the texel with s coordinate tex_s_max:

```
gsSPTexture((tex_s_max)<<6, (tex_t_max)<<6, 0,
                    G_TX_RENDERTILE, G_ON),
```

?

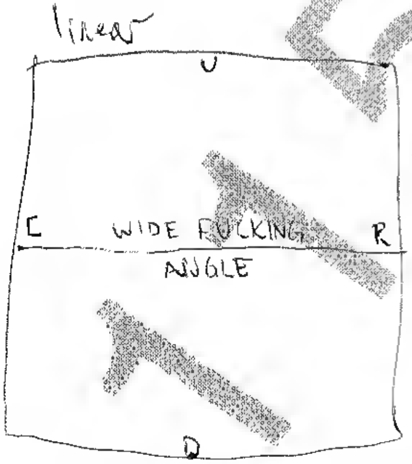
TEX GEN

The left shift of argument by 5 bits is done to account for the S10.5 16-bit internal representation of the texture coordinates (see Texture State below) after a multiplication by one-half in the microcode.



The texture coordinate transformation depends on the geometry mode of the RSP. Two modes are supported, regular and linear.

The first mode (regular) derives the texture coordinates from the x and y projection values, multiplied by the above mentioned scale. In this mode the S coordinate represents the x componant in world coordinates of the direction from the object to the point which should be reflected. The T coordinate represents the Y componant. This means that your texture map should represent the following mapping: 1) The center of the texture map is what is directly behing you. 2) The circle inscribed in the texture map boundaries is what is directly in front of you. 3) The circle with a radius of 0.707 times the radius of the circle in 2) is the objects directly to your left, right, up, down, etc. 4) other points map respectively. what?



The second mode (linear) derives the texture coordinates from the inverse cosine of the x and y projection values, multiplied by the scale. In this mode the S coordinate is the angle of the direction of the reflected vector in the XZ plane. The T coordinate is the angle of the direction in the YZ plane. This mode is useful because you can use a panoramic picture of the horizon for your texture map. The center og the texture map should be the horizon directly behind you. The extremes of the texture map to the left and right should be the horizon in the direction which is directly in front of you. The top of the panoramic texture map should be a constant sky color, and the bottom a constant ground color. When the yaw of the viewing angle changes it is a simple matter to adjust the S position of the texture map so that the new "directly behind" position is the new center of the texture map.

Reflection mapping requires the lighting and texture generation mode bits to be turned on. The first mode (regular) is set using the macro

```
gsSPSetGeometryMode(G_LIGHTING | G_TEXTURE_GEN),
```

while the second mode (linear) is set with

```
gsSPSetGeometryMode(G_LIGHTING| G_TEXTURE_GEN|
G_TEXTURE_GEN_LINEAR),
```

Compatibility with Specular Highlighting

Reflection mapping uses texture mapping so it cannot be used with objects which are otherwise texture mapped. However, reflection mapping can be used in conjunction with one specular highlight. This is analogous to rendering two specular highlights, and utilizes the 2 cycle mode. The specular highlight texture is set for a second tile and accessed in the second cycle. Alternatively, specular highlights can be combined with reflection mapping by incorporating the specular highlights (as bright dots) into the reflection map texture wherever the lights are located. This technique permits an unlimited number of specular highlights.

Environment Mapping

Reflection mapping provides a simple means for carrying out environment mapping. The texture map needs to be an image of the environment as seen from the "viewpoint" of the reflecting object. The main difficulty with this procedure is, of course, generating a suitably realistic texture map.

One simple, yet effective, way to generate an environment map is to first render the scene as viewed by the object. Render all the objects in the scene using a viewing matrix obtained from a guLookAt call where the Eyex, Eyey,Eyez is at the center of the object and Atx, Aty, Atz is at the eyepoint. Render this scene into a 16 bit, 32 pixel x 32 pixel framebuffer which is not part of the main framebuffer. Then re-render the entire scene into the main framebuffer using the previously rendered 32x32 pixel texture map as an environment map for the reflective object. Larger texture maps can be used by playing with tiling. This is not a mathematically perfect way to generate an environment map. but it is relatively cheep, and very effective. Try using different aperature angles in the perspective call while rendering the texture map and turning G_TEXTURE_GEN_LINEAR on or off to tweak the effect.

Vertex Fog State

Fog alters the color of objects based on their distance from the eye position. Fog can be used to make objects blend into the background color as they get farther away. One problem which can be fixed by fog is that when an object goes beyond the far clipping boundary and is clipped away it suddenly dissapears. If fog is enabled the object can be made to look more and more like the background color until, when the object reaches the far clipping plane, the object is exactly the same color as the background and no one notices when it dissappears.

The use of fog requires that the following steps be taken:

1) run in two cycle mode.

2) Set the render mode to blend the fog color with the primitive color.

3) Set the fog position.

4) Enable fog.

5) Set the Fog Color.

For example:

```
/* 2 cycle mode */
gsDPSetCycleType(G_CYC_2CYCLE),
/* blend fog in AA ZB mode */
gsDPSetRenderMode(G_RM_FOG_SHADE_A,G_RM_AA_ZB_OPA_SURF2),
/* set fog position and enable fog */
gsSPFogPosition(FOG_MIN, FOG_MAX)
gsSPSetGeometryMode(G_FOG),
/* set the fog color */
gsDPSetFogColor(RED,GREEN,BLUE, ALPHA),
```

FOG_MIN specifies the position where fog begins and FOG_MAX represents where fog is thickest. Both values are integers and are mapped linearly such that 0={at the near clipping plane}, and 1000={at the far clipping plane}. FOG_MAX is generally set to 1000 so that objects are completely "fogged out" when they hit the far plane, but not before then. FOG_MIN is set to the position where fog starts. A value of 0 will make the object slowly change to fog color as it retreats from the viewer, while a larger value (eg 800) will make the object clearly visible until it gets 80% of the way to the far plane where it will finally begin to "fog out." Note that perspective makes distant objects look *much* farther away than nearby objects. Because of this some objects which don't appear to be very far away may be more affected by fog than expected even though the FOG_MIN value is fairly high. To remedy this problem simply increase the FOG_MIN value until you get the desired effect. For example if you set FOG_MIN to 500, but objects which are about midway between the far and near planes look foggier than they should, just increase the value of FOG_MIN until they look better.

Fog works well when the horizon is a constant color (the same as the fog color). When the horizon color is complicated (eg clouds, gradient colors, etc), you can make objects become transparent when they are distant. To do this don't set the G_RM_FOG_SHADE_A render mode or the Fog color. Just enable fog, use a transparent render mode, and swap FOG_MAX and FOG_MIN. FOG_MIN should be set to 1000 to make the object completely transparent when it is at the far clipping plane. FOG_MAX should be a large enough value that fog has no effect until the object is farther away than any other objects are likely to be (ie beyond mountains and other terrain, etc.). Because transparency is used, the z-buffer will not keep things behind the transparent-fogged object from being hidden, so it should only be enabled for objects which are already fairly far from the viewer. This special transparent-fog mode should be used with caution (as compared with the regular fog effect described in the preceding paragraphs which should work consistantly).

Fog is independant of lighting and texture mapping so it may be used in conjunction with any, all, or none of these other effects.

11549

Primitives

Availability of different geometry primitives depends on the version of the RSP microcode which has been loaded for execution.

Triangles

Table 12-11 gsSP1Triangle(int v0, int v1, int v2, int flag)

Parameter	Values
v0	vertex buffer index of the first coordinate. (0-15)
v1	vertex buffer index of the second coordinate. (0-15)
v2	vertex buffer index of the third coordinate. (0-15)
flag	used for flat shading; ordinal id of the vertex parameter to use for shading: 0, 1, or 2

Other bits of the flag field are currently reserved.

Lines

Table 12-12 gsSPLine3D(int v0, int v1, int flag)

Parameter	Values
v0	vertex buffer index of the first coordinate. (0-15)
v1	vertex buffer index of the second coordinate. (0-15)
flag	unused (should be 0)

Lines are only available when running the line microcode. All the normal vertex attributes (color, texture, z) are also available for lines. Lines however require separate rdp rendermodes to be set than for polygons. Consult the man pages for more details. Z-buffered lines will only do reads of the z-buffer, and not writes. Thus z-buffered lines should be drawn after z-buffered polygons.

Rectangles

All rectangles are 2D primitives, specified in screen-coordinates. They are not clipped, but they are scissored in a limited fashion. In 1CYCLE and

1151389

2CYCLE mode, rectangles are scissored in the same way as triangles. In COPY and FILL modes, rectangles are scissored to four pixel boundaries; meaning that additional scissoring may be necessary in the application program.

Filled rectangles are implemented entirely in the RDP, as "pass-through" commands with respect to the RSP. They are mentioned here for completeness:

Table 12-13 gsDPFillRectangle(unsigned int ulx, unsigned int uly, unsigned int lrx, unsigned int lry)

Parameter	Values
ulx	screen coordinate of upper-left x (10.2 format)
uly	screen coordinate of upper-left y (10.2 format)
lrx	screen coordinate of lower-right x (10.2 format)
lry	screen coordinate of lower-right y (10.2 format)

Textured rectangles require minimal RSP intervention, and are thus an SP operation:

Table 12-14 gsSPTextureRectangle(unsigned int ulx, unsigned int uly, unsigned int lrx, unsigned int lry, int tile, short int s, short int t, short int dsdx, short int dtdy)

Parameter	Values
ulx	screen coordinate of upper-left x (10.2 format)
uly	screen coordinate of upper-left y (10.2 format)
lrx	screen coordinate of lower-right x (10.2 format)
lry	screen coordinate of lower-right y (10.2 format)
tile	which tile in TMEM to use
s	s coordinate of upper-left corner (S10.5 format)
t	t coordinate of upper-left corner (S10.5 format)
dsdx	change in s per change in x coordinate (S5.10 format)
dtdy	change in t per change in y coordinate (S5.10 format)

There is a related macro, *gsSPTextureRectangleFlip()*, that is identical to *gsSPTextureRectangle()*, except that the texture is flipped so that the s

1157309

coordinate changes in the y direction, and the t coordinate changes in the x direction:

Table 12-15 gsSPTextureRectangleFlip(unsigned int ulx, unsigned int uly, unsigned int lrx, unsigned int lry, int tile, short int s, short int t, short int dtdx, short int dsdy)

Parameter	Values
ulx	screen coordinate of upper-left x (10.2 format)
uly	screen coordinate of upper-left y (10.2 format)
lrx	screen coordinate of lower-right x (10.2 format)
lry	screen coordinate of lower-right y (10.2 format)
tile	which tile in TMEM to use
s	s coordinate of upper-left corner (S10.5 format)
t	t coordinate of upper-left corner (S10.5 format)
dtdx	change in t per change in x coordinate (S5.10 format)
dsdy	change in s per change in y coordinate (S5.10 format)

1157389

Controlling the RDP State

The RSP performs two functions to support programming the RDP: *segmented address fix-up* and handling *setothermode.*

Segmented address fix-up. Since the RDP is a physical address machine, the RSP must translate the segmented addresses present in the display list into physical addresses for the RDP. It does so by filtering out any RDP command with an address (the 'set image' commands) and patching the address before passing it to the RDP.

The RDP *setothermode* register is a collection of state bits, affecting many different functions of the RDP. In order to simplify programming the RDP state, the RSP caches the SETOTHERMODE command, and presents a simpler "set/clear" interface through the display list. See Chapter 13, "RDP Programming" for more details of these macros.

1157389

Chapter 13

RDP Programming

115189

The Reality Display Processor (RDP) rasterizes triangles and rectangles, and produces high-quality, Silicon Graphics style pixels that are textured, antialiased, and z-buffered.

The RDP has four main configurations where all the individual blocks work together to generate pixels. These main configurations are called "cycle types," because they indicate how many pixels are generated per cycle. The following table indicates their peak performance. Keep in mind that these peak numbers are typically realized on large rectangle primitives. Triangles have variable short and long spans and these numbers degrade rapidly. The following table lists the RDP's performance.

Table 13-1 Cycle Types

Type	Performance
FILL	4 16 bit pixels/cycle 2 32 bit pixels/cycle
COPY	4 pixels/cycle
1CYCLE	1 pixel/cycle
2CYCLE	1 pixel/2 cycles

Note: These are theoritical peak performances. In reality, due the memory latency and buffering overhead, actual performance numbers are lower.

RDP Pipeline Blocks

The RSP performs 3D geometric transformations while the RDP pipeline rasterizes the polygon. The RDP consist of several pipeline subblocks. There are six major logical RDP blocks: the RS, TX, TF, CC, BL, and MI. The connections between these blocks can be reconfigured to the four cycle types listed in Table 13-1, to perform different rasterization operations.

Table 13-2 Basic Operations of RDP Subblocks

Block	Functionality
RS	The RaSterizer generates pixel coordinates and their attributes' slopes. Pixel coordinates consist of X and Y. Attributes consist of R, G, B, A, Z, S/W, T/W, 1/W, L, pixel coverage.
TX	The TeXturing unit contains texture memory and samples the texture, based on which texel represents the pixel being processed in the pipeline.
TF	The Texture Filter performs a 4-to-1 bilinear filter of 4 texel samples to produce a single bilinear filtered texel.
CC	The Color Combiner performs general blending of color sources by linearly interpolating between two colors with a coefficient. For example, it may take the filtered texel samples and the shading color (RGBA) and combine them together.
BL	The BLender blends the pipeline-processed pixels with the pixels in the framebuffer. The blender can do transparencies and also sophisticated antialiasing operations.
MI	The Memory Interface performs the actual read/modify/write cycles to and from the framebuffer.

Note: The six RDP blocks (RS, TX, TF, CC, BL, and MI) are purely logical blocks. For example, the hardware implementation of RS consist of several blocks. However, for programming, each can be treated as a single logical block.

1157369

One-Cycle-per-Pixel Mode

The pipeline configuration illustrated in Figure 13-1 shows how the RDP blocks are connected in one-cycle-per-pixel mode.

Figure 13-1 One-Cycle Mode RDP Pipeline Configuration

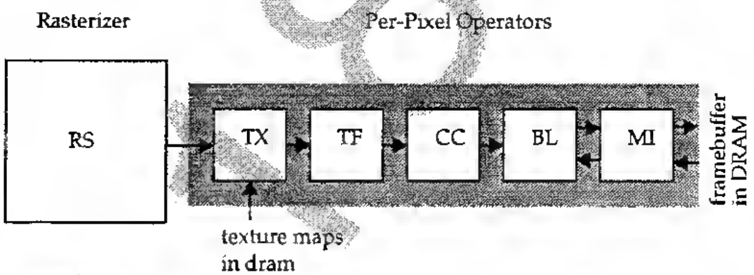


Table 13-3RDP Pipeline Block Functionality in One-Cycle Mode

Block	Functionality
RS	Generates pixel and its attribute covered by the interior of the primitive.
TX	Generates 4 texels nearest to this pixel in a texture map.
TF	Bilinear filters 4 texels into 1 texel, OR performs step 1 of YUV-to-RGB conversion.
CC	Combines various colors into a single color, OR performs step 2 of YUV-to-RGB conversion.
BL	Blends the pixel with framebuffer memory pixel, OR fogs the pixel for writing to framebuffer.
MI	Fetches and writes pixels from and to the framebuffer memory.

One-cycle mode fills a fairly high-quality pixel. You can generate pixels that are perspectively corrected, bilinear filtered, modulate/decal textured, transparent, and z-buffered, at one-cycle-per-pixel peak bandwidth.

1157389

Note: Reaching peak bandwidth is difficult. The framebuffer memory is organized in row order. In small triangles, it is rare to have long horizontal runs of pixels on a single scanline. In these cases, the pipeline is often stalled, pending memory access for read or write cycles.

Two-Cycles-per-Pixel Mode

The RDP blocks can be reconfigured into a two-cycle-per-pixel pipeline structure for additional functionality. Figure 13-2 shows the RDP pipeline in 2-cycle mode where one pixel is generated every 2 clocks.

Figure 13-2 Two Cycle Mode RDP Pipeline configuration

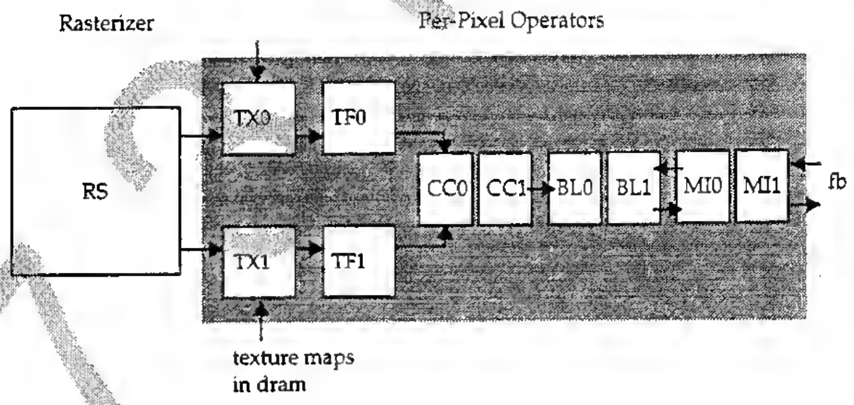


Table 13-4RDP Pipeline Block Functionality for Two-Cycle Mode

Block	Functionality
RS	Generates a pixel and its attribute covered by the interior of the primitive.
TX0	Generates 4 texels nearest to this pixel in a texture map. This can be level X of a mipmap.
TX1	Generates 4 texels nearest to this pixel in a texture map. This can be level X+1 of a mipmap.
TF0	Bilinear; filters 4 texels into 1 texel.

115789

Table 13-4RDP Pipeline Block Functionality for Two-Cycle Mode

Block	Functionality
TF1	Bilinear; filters 4 texels into 1 texel, OR step 1 of YUV-to-RGB conversion.
CC0	Combines various colors into a single color, OR linear interpolates the 2 bilinear filtered texels from 2 adjacent levels of a mipmap, OR performs step 2 of YUV-to-RGB conversion.
CC1	Combines various colors into a single color, OR chroma keying.
BL0	Combines fog color with resultant CC1 color.
BL1	Blends the pipeline pixels with framebuffer memory pixels.
MI0	Read/modify/write color memory.
MI1	Read/modify/write Z memory.

Two-cycles-per-pixel mode contains more features than one-cycl- per-pixel mode. In addition to all of the features of one-cycle mode, two-cycle mode can also do mipmapping and fog.

Note: MI0 and MI1 represent two cycles of the MI that access color and z framebuffer cycles, respectively. This is only a logical representation. The MI does not need to run two cycles to do color and z-buffer access. One cycle per pixel mode can also perform color and z-buffer accesses. The reason for this representation is to show that two MI access cycles are balanced in the two-cycle mode. In one-cycle mode, the pipeline is often stalled at MI, waiting for the framebuffer when accessing both color and z.

These RDP blocks are very flexible and can be configured to do many things. Table 13-4 outlines the typical usage of these blocks for a powerful rasterization pipeline. Study the following sections to understand what attribute state is programmable within each RDP block to master the raster subsystem.

Fill Mode

For high-performance framebuffer clearing, the RDP has a fill mode, which can fill 64 bits per clock. A programmable RDP color attribute is written into the framebuffer during each 64-bit write cycle. The RDP arithmetic pipeline is largely unused, because the computation can not keep up with the pixel fill rate. The fill mode is most commonly used for clearing color and z-buffers.

Note: In fill mode, use the render mode g*DPSetRenderMode(G_RM_NOOP, G_RM_NOOP2) to put the blender into a safe state. Attempting to read Z when in fill mode can cause the RDP pipeline to hang.

Copy Mode

For high-performance image-to-image copies, RDP also supports a copy mode that can copy 64 bits or 4 pixels per clock. The RDP texture memory in the TX is just a buffer capable of holding up to 4 KB worth of image pixels. You can load bitmaps into this buffer as well as writing back out to the framebuffer. The is a common bit blit operation that many 2D graphics hardware systems support. Once again, the RDP arithmetic pipeline is largely unused in copy mode.

Note: One important operation that does work in copy mode is alpha compare. This allows RDP to blit an image into the framebuffer and conditionally remove image pixels with alpha = 0. Usually, images with alpha = 0 represent transparency, see "Alpha Compare Calculation" on page 315 for more details.

Note: In copy mode, use the render mode g*DPSetRenderMode(G_RM_NOOP, G_RM_NOOP2) to put the blender into a safe state.

1153189

RDP Global State

Several state are global to the RDP, usually to specify pipeline configuration and synchronization.

Cycle Type

To configure the pipeline for rendering, choose one of the cycle types that offers the functionality required at peak performance.

Table 13-5 gsDPSetCycleType(type)

Parameter	Values
type	G_CYC_1CYCLE G_CYC_2CYCLE G_CYC_COPY G_CYC_FILL

115713789

Synchronization

You might ask "How does the primitive rendering pipeline synchronize with all of the different attribute states that the programmer can set?" Imagine that the last few pixels are being processed in the RDP pipeline when it receives a new attribute command, and this command affects the pixel currently being processed. You would not want the last few pixels of a primitive to have the attributes of a following primitive. You really want to have the attribute state only to modify the pixels of the primitive following the attribute state change. This synchronization is not implicit within the pipeline; the application must explicitly insert proper synchronization between attribute state changes and primitives.

Table 13-6 gsDPPipeSync()

Parameter	Values
none	none

This command synchronizes the attribute update with respect to primitive rendering. It ensures that the last pixels of a primitive are rendered prior to the attribute taking effect. Insert this inbetween an RDP primitive followed by an RDP attribute:

```
gDPSetCycleType(glistp++, G_CYC_FILL);
gDPFillRectangle(glistp++, 0, 0, 127, 127);
gDPPipeSync(glistp++);
gDPSetCycleType(glistp++, G_CYC_1CYCLE);
```

Note: After a primitive (eg. gSPTriangle, gDPFillRectangle, gDPTextureRectangle) and before an RDP attributes (eg. gDPSet*), you need to insert a gDPPipeSync.

After processing all of the RDP display list, the host processor must be interrupted and notified.

Table 13-7gsDPFullSync()

Parameter	Value
none	none

gDPFullSync() also shuts down the RDP until given a new DP DL to eliminate excessive power consumption.

Span Buffer Coherency

For RMW cycles, the RDP is smart enough to prefetch a row of pixels as soon as the X, Y coordinates of the span are determined. The RDP then preloads the framebuffer content of this span into an RDP onchip span buffer. The RDP then waits for the pipeline to process the parameters for the outgoing pixels. When the outgoing pixels are computed, they are "combined" with the preloaded framebuffer pixels before writing back to the framebuffer.

An example of this operation is z-buffer and transparency blending. (This is not shown in the logical pipeline description earlier, to simplify the understanding of the pipeline.)

11513199

The RDP has enough onchip RAM to hold several span buffers. Therefore, what would happen if two spans in sequence happened to overlap the same screen area? The RDP would prefetch the first span into a span buffer while the pipeline starts processing this span. Then it would prefetch the next span into another span buffer.

This is where the problems occur: the pixel data for the next span is not yet computed. The RDP does have span buffer coherency, at the cost of some performance. If errors are objectionable in your animation, use gsDPPipelineMode(G_PM_1PRIMITIVE) to cause all primitives to add between 30 to 40 null cycles after the last span of a primitive is rendered.

Table 13-8gsDPPipelineMode(mode)

Parameter	Value
mode	G_PM_1PRIMITIVE G_PM_NPRIMITIVE

These dead cycles can be expensive in terms of fill rate so it is recommended not to use the 1PRIMITIVE mode be used unless absolutely necessary.

1157389

RS: Rasterizer

The Rasterizer's main job is implied in its name: to generate pixels that cover the interior of the primitive. The primitives are either triangles or rectangles. For each pixel, the RS generates the following attributes:

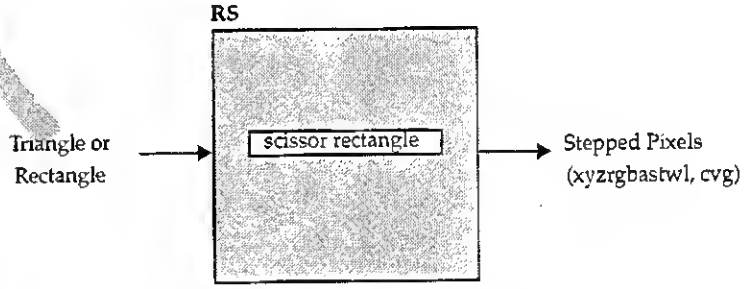
- screen x, y location
- z depth for z-buffer purposes
- RGBA color information
- s/w, t/w, 1/w, lod for texture index, perspective correction, and mipmapping.

  These are commonly referred to as s, t, w, l.
- coverage value.

  Pixels on the edge of primitives have partial coverage values. Interiors are full.

These values are sent to the pipelined blocks downstream for other computations, such as texture sampling, color blending, and so on.

Figure 13-3 RS State and Input/Output



Scissoring

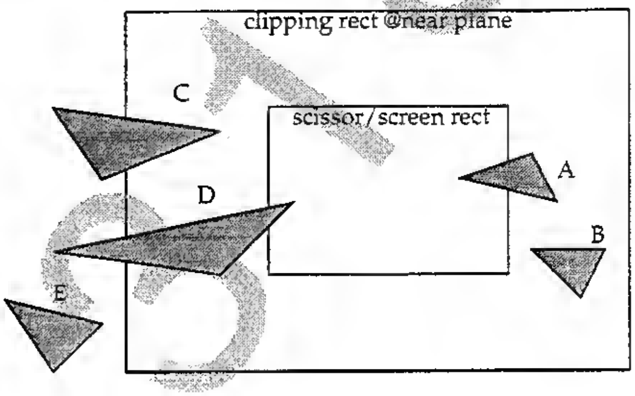
Scissoring is commonly used to eliminate running performance-intensive clipping code in the geometry processing stage of a graphics pipeline. You do this by projecting the clipping rectangle at the near plane larger than the

11573189

scissor rectangle. The rasterizer can then efficiently eliminate the portion outside of the screen rectangle.

The RSP geometry processing is performed in fixed-point arithmetic. The clipped rectangle boundary is not a perfect rectangle, because of precision errors. This artifact can also be eliminated using the scissoring rectangle.

Figure 13-4 Scissor/Clipping/Screen Rectangles



Triangle A is scissored, but not clipped. B, C and E are trivially rejected because no pixels are enumerated. Only D is clipped and scissored.

Table 13-9 gsDPSetScissor(ulx, uly, lrx, lry)

Parameter	Value
ulx	upper left x
uly	upper left y
lrx	lower right x
lry	lower right y

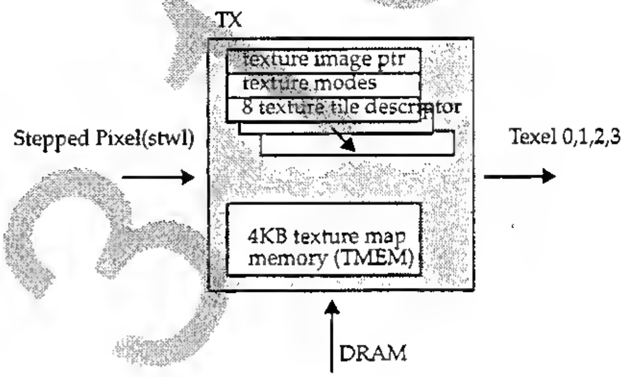
Note: Rectangles are scissored with some restrictions. In 1CYCLE and 2CYCLE mode, rectangles are scissored the same as triangles. In FILL and COPY mode, rectangles are scissored to the nearest four pixel boundary; this might require rectangles to be scissored in screen space by the game software.

11573189

TX: Texture Engine

The Texture Engine takes s/w, t/w, 1/w, and lod values for a pixel and fetches the onboard texture memory for the four nearest texels to the screen pixel. The game application can manipulate TX states such as texture image types and formats, how and where to load texture images, and texture sampling attributes.

Figure 13-5 TX State and Input/Output


TX
texture image ptr
texture modes
8 texture tile descriptor
Stepped Pixel(stwl)
Texel 0,1,2,3
4KB texture map memory (TMEM)
DRAM

Texture Tiles

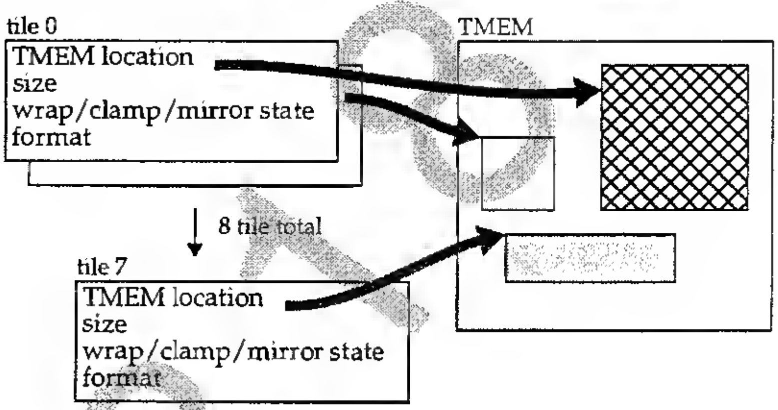
TX treats the 4 KB on-chip texture memory (TMEM) as general-purpose texture memory. The texture memory is divided into four simultaneous accessible banks, giving output of four texels per clock.

The game application can load varying-sized textures with different formats anywhere in the 4 KB texture map. There are eight texture tile descriptors that describe the location of texture images within the TMEM, the format of this texture, and the sampling parameters. Therefore, you can load many

1157389

texture maps in the TMEM at one time, but there are only eight tiles that are accessible at any time.

Figure 13-6 Tile Descriptors and TMEM


tile 0
TMEM location
size
wrap/clamp/mirror state
format
8 tile total
tile 7
TMEM location
size
wrap/clamp/mirror state
format
TMEM

Note: There are some restrictions, depending on texel size and 64-bit alignment within the texture memory.See "Alignment" on page 259.

Multiple Tile Textures

Given the eight texture tiles, you can use two- cycle pipeline mode to cycle TX twice and access eight texels (four from each of two tiles). This functionality, coupled with the use of up to eight texture tiles, allows the TX to perform mipmapping and detailed textures.

Furthermore, there are no explicit restrictions requiring power of two tile-sized decrements for mipmaps. Multi-tile texture map sizes are all independently programmable. Therefore, using these tiles and the color combiner block (see Chapter 13, "CC: Color Combiner"), arithmetic logic can result in many special effects. For example, sliding two different frequency band tiles across a polygon surface while combining them with a blue polygon can give a nice ocean wave effect.

Texture Image Types and Format

Table 13-10 shows the legal combinations of data types and pixel/texel sizes for the Color and Texture images. For RGBA types, the 16-bit format is 5/5/5/1, and the 32-bit format is 8/8/8/8.

The Intensity Alpha type (IA) replicates the I value on the RGB channels and places the A value on the A channel. The IA 16-bit format is 8/8, the 8-bit format is 4/4, and the 4-bit format is 3/1.

Table 13-10 Texture Format and Sizes

Type	4b	8b	16b	32b
RGBA			X	X
YUV			X	
Color Index	X	X		
IA	X	X	X	
I	X	X		

Texture Loading

Several steps are necessary to load a texture map into the TMEM. You must block-load the texture map itself and set up the attributes for this tile. There are GBI macros that simplify all these steps into a single macro.

There are two ways of loading textures: block or tile mode. Block mode assumes that the texture map is a contiguous block of texels that represents the whole texture map. Tile mode can lift a subrectangle out of a larger

11573169

image. The following tables list block and tile mode texture-loading GBI commands respectively.

Table 13-11 gsDPLoadTextureTile(timg, fmt, siz, width, height, uls, ult, lrs, lrt, pal, cms, cmt, masks, maskt, shifts, shiftt)

Table 13-12 gsDPLoadTextureTile_4b(pkt, timg, fmt, width, height, uls, ult, lrs, lrt, pal, cms, cmt, masks, maskt, shifts, shiftt)

Parameter	Value
timg	Texture dram address.
fmt	G_IM_FMT_RGBA G_IM_FMT_YUV G_IM_FMT_CI G_IM_FMT_I G_IM_FMT_IA
siz	G_IM_SIZ_4b G_IM_SIZ_8b G_IM_SIZ_16b G_IM_SIZ_32b
width, height	Texture tile width and height in texel space.
pal	TLUT palette.
cms, cmt	clamping/mirroring for s/t axis G_TX_NOMIRROR G_TX_MIRROR G_TX_WRAP G_TX_CLAMP
masks, maskt	Bit mask for wrapping. G_TX_NOMASK or a number: A wrapping bit mask is represented by (1<<number) - 1.

11573199

Table 13-11 gsDPLoadTextureTile(timg, fmt, siz, width, height, uls, ult, lrs, lrt, pal, cms, cmt, masks, maskt, shifts, shiftt)

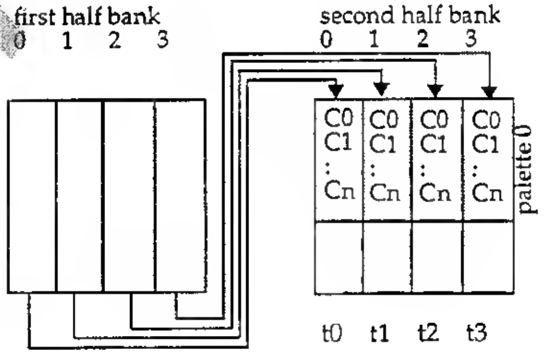
Table 13-12 gsDPLoadTextureTile_4b(pkt, timg, fmt, width, height, uls, ult, lrs, lrt, pal, cms, cmt, masks, maskt, shifts, shiftt)

Parameter	Value
shifts, shiftt	Shifts applied to s/t coordinate of each pixel. This is how you "sample" the lower levels of a mipmap. G_TX_NOLOD or a number: (s or t coord >> number) = s/t to sample other mipmap levels.
uls	upper left s index of the tile within the texture image
ult	upper left t
lrs	lower right s
lrt	lower right t

Color-Indexed Textures

There are some restrictions on the size and placement of CI texture maps within the TMEM. The TMEM is actually partitioned into two halves. Four texels are sampled from the first bank and fed into the second bank for texture/color/index table lookup (TLUT).

Figure 13-7 CI TMEM Partition



Four texels from the texture images are sent from first half banks to the second half banks. The second half banks contain color index palettes. Each

11513109

color map entry is replicated 4 times for four simultaneous bank lookups. Therefore, 8-bit CI textures all require 2 KB (256 x 64 bits per entry) second half banks to hold the TLUT, while 4-bit CI texture can have up to 16 separate TLUTs.

Note: TLUT must reside on the second half of TMEM; while CI texture cannot reside on the second half of TMEM. Non-CI texture can actually reside on the second half of TMEM in unused TLUT palette/entries.

Table 13-13gsLoadTLUT(count, tmemaddr, dramaddr)

Parameter	Value
count	Number of entries in the TLUT. For example, 4-bit texel TLUT would have 16 entries.
tmemaddr	Where the TLUT goes in TMEM.
dramaddr	Where the TLUT is in DRAM.

Texture-Sampling Modes

Software can enable and disable TX to perform the follow sampling modes:

- perspective correction
- detail or sharpen textures
- LOD (mipmap) or bilinear textures
- RGBA or IA TLUT type.

Table 13-14gsDPSetTexturePersp(mode)

Parameter	Value
mode	G_TP_NONE G_TP_PERSP

1157389

Table 13-15 gsDPSetTextureDetail(mode)

Parameter	Value
mode	G_TD_CLAMP G_TD_SHARPEN G_TD_DETAIL

Table 13-16 gsDPSetTextureLOD(mode)

Parameter	Value
mode	G_TL_TILE G_TL_LOD

Table 13-17 gsSetTextureLUT(type)

Parameter	Value
type	G_TT_NONE G_TT_RGBA16 G_TT_IA16

Synchronization

With TMEM and tile descriptor states, TX also requires explicit synchronization to render primitives with the proper attribute state. Texture loads after primitive rendering must be preceded by a gsDPLoadSync(), and tile descriptor attribute changes should be preceded by a gsDPTileSync().

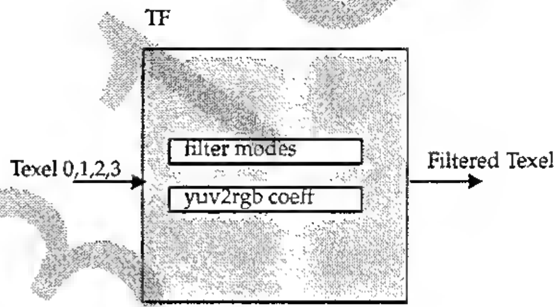
Note: If you use the high-level programming macros gsDPLoadTexture* or gsDPLoadTexture*_4b, then you don't need to worry about load and tile syncs. They are embedded in the macro.

11571819

TF: Texture Filter

Texture filter takes the four texels generated by TX and produces a simple bilinear-filtered texel. The TF can also work together with the color combiner (see Chapter 13, "CC: Color Combiner") to perform YUV-to-RGB color space conversion.

Figure 13-8 Texture Filter State and Input/Output



Filter Types

TF performs three types of filter operations: point sampling, box filter, and bilinear interpolation. Point sampling just selects the nearest texel to the screen pixel. In the special case where the screen pixel is always the center of four texels, the box filter can be used. In a typical 3D, arbitrarily rotated polygon, the bilinear filter is the best choice available.

Note: For hardware cost reduction, the RDP does not implement a true bilinear filter. Instead, the three nearest texels are linearly interpolated to produce the result pixels. This has a natural triangulation bias. This artifact is not noticeable in normal texture images. However, in regular pattern

11573189

images, it can be noticed. For example, notches can be seen in the crosshair on a image of grids. This can be eliminated by prefiltering the image with a wider filter.

Table 13-18 gsSetTextureFilter(type)

Parameter	Value
type	G_TF_POINT G_TF_AVERAGE G_TF_BILERP

Color Space Conversion

Color space conversion can be used to convert YUV textures into RGB. This could be a useful compression technique, or it could be used for MPEG video, or for special effects.

Table 13-19 gsSetTextureConvert(mode)

Parameter	Value
mode	G_TF_CONV G_TF_FILTCONV G_TF_FILT

Table 13-20 gsSetConvert(k0,k1,k2,k3,k4,k5)

Parameters	Value
k0, k1, k2	G_CV_K0, G_CV_K1, G_CV_K2
k3, k4, k5	G_CV_K3, G_CV_K4, G_CV_K5

Note: The default state of the RDP is G_TF_CONV (perform YUV2RGB), which is probably not what you want (if you are using RGB textures). A common bug is to forget to set this (usually it should be G_TF_FILT).

11573189

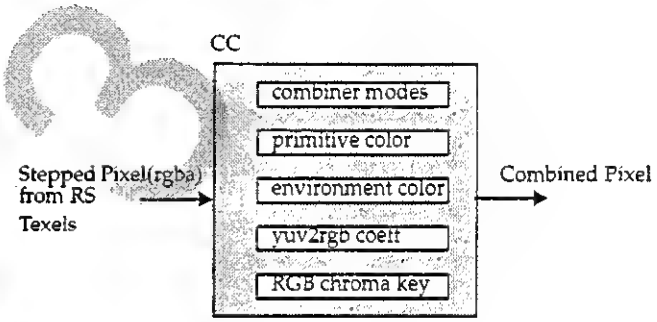
CC: Color Combiner

The color combiner (CC) combines texels from TX and stepped RGBA pixel values from RS. The CC is the ultimate paint mixer. It can take two color values from many sources and linearly interpolate between them. The CC basically performs this equation:

$$newcolor = (A - B) \times C + D$$

Here, A, B, C, and D can come from many different sources. Notice that if D=B, then this is a simple linear interpolator.

Figure 13-9 Color Combiner State and Input/Output


CC
combiner modes
primitive color
environment color
yuv2rgb coeff
RGB chroma key
Stepped Pixel(rgba) from RS
Texels
Combined Pixel

Most of CC programming involves setting the desired sources for (A,B,C,D) of the equation above. There are also programmable color registers within CC that can be used to source (A,B,C,D) input of the interpolator.

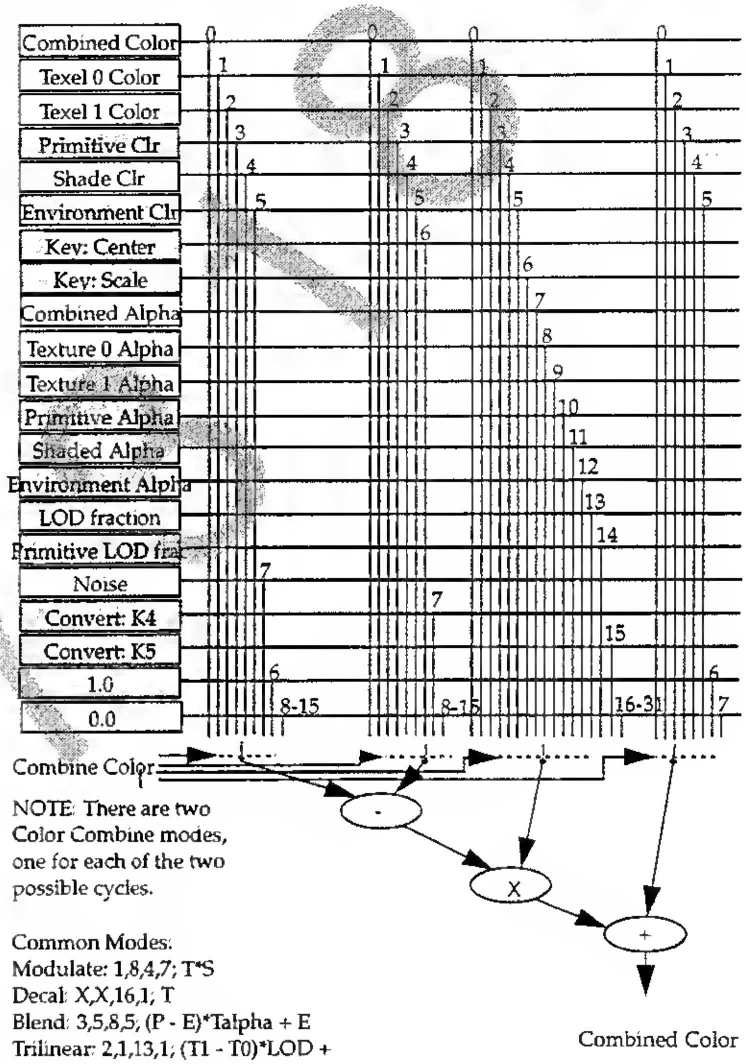
Color and Alpha Combiner Inputs Sources

The following picture describes all possible input selection of a general purpose linear interpolator for RGB and Alpha color combination.The input

1157389

in the shaded boxes are CC internal state that you can set. Most are programmable color registers.

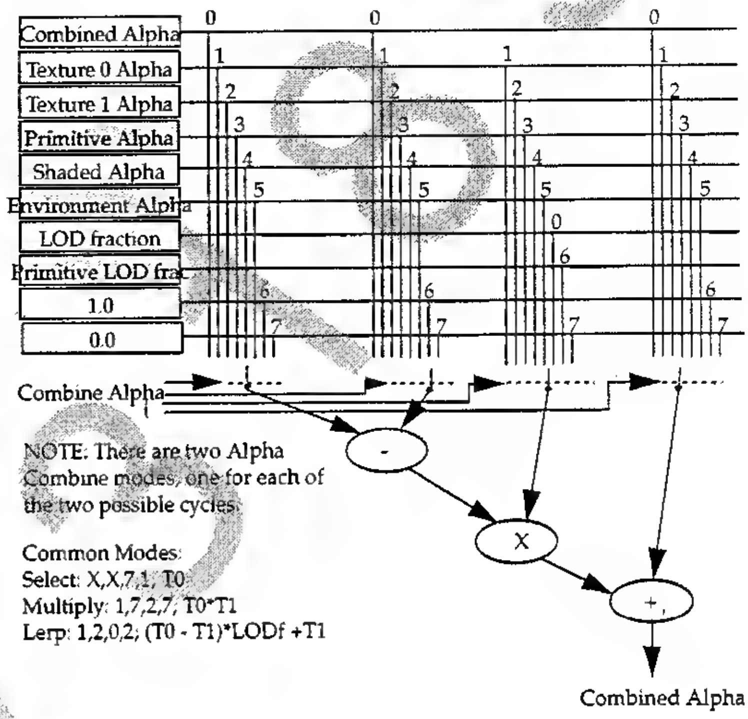
Figure 13-10 RGB Color Combiner Input Selection


Combined Color
Texel 0 Color
Texel 1 Color
Primitive Clr
Shade Clr
Environment Clr
Key: Center
Key: Scale
Combined Alpha
Texture 0 Alpha
Texture 1 Alpha
Primitive Alpha
Shaded Alpha
Environment Alpha
LOD fraction
Primitive LOD frac
Noise
Convert: K4
Convert: K5
1.0
0.0
8-15
8-15
16-31
Combine Color
-
X
+
Combined Color

NOTE: There are two Color Combine modes, one for each of the two possible cycles.

Common Modes:
Modulate: 1,8,4,7; T*S
Decal: X,X,16,1; T
Blend: 3,5,8,5; (P - E)*Talpha + E
Trilinear: 2,1,13,1; (T1 - T0)*LOD + T0
Interference: 1,8,2,7; T0 * T1
Keying:1,6,6,7; (T0 - Center) * Scale + 0

Figure 13-11 Alpha Combiner Input Selection


Combined Alpha
Texture 0 Alpha
Texture 1 Alpha
Primitive Alpha
Shaded Alpha
Environment Alpha
LOD fraction
Primitive LOD frac
1.0
0.0
0
1
2
3
4
5
6
7
Combine Alpha
-
X
+
Combined Alpha

NOTE: There are two Alpha Combine modes, one for each of the two possible cycles.

Common Modes:
Select: X,X,7,1; T0
Multiply: 1,7,2,7; T0*T1
Lerp: 1,2,0,2; (T0 - T1)*LODf +T1

CC Internal Color Registers

There are two internal color registers in the CC: primitive and environment color. The primitive color can be used to set a constant polygon face color. The environment color can be used to represent the ambient color of the environment. Both can be used as source for linear interpolation. The names "primitive" and "environment" are purely arbitrary; you can use them for any purpose you wish.

Table 13-21 gsSetPrimColor(minlevel, frac, r, g, b, a), gsDPSetEnvColor(r, g, b, a)

Parameter	Value
minlevel	minimum LOD level
frac	LOD fraction for blending two texture tiles
r, g, b, a	color

One-Cycle Mode

Many of the typical RGB and alpha input selections are predefined in Table 13-24. In 1 cycle mode bothe mode1 and mode2 should be the same. See the man page for gDPSetCombineMode for a description of each mode setting.

Table 13-22 One-Cycle Mode Using gsDPSetCombineMode(mode1, mode2)

Parameter	Value
mode1/2	G_CC_PRIMITIVE G_CC_SHADE G_CC_ADDRGB G_CC_ADDRGBDECALA G_CC_SHADEDECALA
mode1/2	Decal textures in RGB, RGBA formats G_CC_DECALRGB G_CC_DECALRGBA

115713189

Table 13-22 One-Cycle Mode Using gsDPSetCombineMode(mode1, mode2)

Parameter	Value
mode1/2	Modulate texture in I, IA, RGB, RGBA formats G_CC_MODULATEI G_CC_MODULATEIA G_CC_MODULATEIDECALA G_CC_MODULATERGB G_CC_MODULATERGBA G_CC_MODULATERGBDECALA G_CC_MODULATEI_PRIM G_CC_MODULATEIA_PRIM G_CC_MODULATEIDECALA_PRIM G_CC_MODULATERGB_PRIM G_CC_MODULATERGBA_PRIM G_CC_MODULATERGBDECALA_PRIM
mode1/2	Blend texture in I, IA, RGB, RGBA formats. G_CC_BLENDI G_CC_BLENDIA G_CC_BLENDIDECALA G_CC_BLENDRGBA G_CC_BLENDRGBDECALA
mode1/2	Reflection and specular hilite in RGB, RGBA formats. G_CC_REFLECTRGB G_CC_REFLECTRGBDECALA G_CC_HILITERGB G_CC_HILITERGBA G_CC_HILITERGBDECALA

Note: In one-cycle mode, mode1 and mode2 should be the same value.

1157389

Two-Cycle Mode

Color Combiner (CC) can perform two linear interpolation arithmetic computations in two-cycle pipeline mode. Typically, the second cycle is used to perform texture and shading color modulation (in other words, all those modes you saw in one-cycle mode). However, the first cycle can be used for another linear interpolation calculation; for example, LOD interpolation between the two bilinear filtered texels from two mipmap tiles.

Table 13-23 Two-Cycle Mode Using gsDPSetCombineMode(mode1, mode2)

Parameter	Value
mode1	G_CC_TRILERP G_CC_INTERFERENCE
mode2	G_CC_PASS2 Most of the Decal, Modulate, Blend and Reflection/Hilite texture modes mentioned in one cycle mode. However, since they are values for mode2 parameter, the names must all end with 2. e.g. G_CC_MODULATEI2.

Custom Modes

Color Combiner (CC) can be programmed more specifically when you design your own color combine modes. To define a new mode use the format:

#define G_CC_MYNEWMODE a,b,c,d, A,B,C,D

Where the color output will be (a-b)*c+d and the alpha output will be (A-B)*C+D. The values you can use for each of a, b, c, d, A, B, C, and D are:

`COMBINED`	`combined output from cycle 1 mode`
`TEXEL0`	`texture map output`
`TEXEL1`	`texture map output from tile+1`
`PRIMITIVE`	`PrimColor`
`SHADE`	`Shade color`
`ENVIRONMENT`	`Environment color`
`CENTER`	`chroma key center value`
`SCALE`	`chroma key scale value`

COMBINED_ALPHA	combined alpha output from cycle 1
TEXEL0_ALPHA	texture map alpha
TEXEL1_ALPHA	texture map alpha from tile+1
PRIMITIVE_ALPHA	PrimColor Alpha
SHADE_ALPHA	Shade alpha
ENV_ALPHA	Environment color alpha
LOD_FRACTION	LOD fraction
PRIM_LOD_FRAC	Prim LOD fraction
NOISE	noise (random)
K4	color convert constant K4
K5	color convert constant k5
1	1.0
0	0.0

Then you can use your new mode just like a regular mode:

gDPSetCombineMode(G_CC_MYNEWMODE, G_CC_MYNEWMODE);

Chroma Key

The color combiner can be used to perform "chroma keying", which is a process where areas of a certain color are taken out and replaced with a texture. This is a similar effect to "blue screen photography", or as seen on the television news weather maps.

The theory is quite simple; a key color is provided, and all pixels of this color are replaced by the texel color requested. The key color is actually specified as a center and width, allowing soft-edge chroma keying (for blended colors):

Figure 13-12 Chroma Key Equations

```
KeyR = clamp(0, (-abs((R - RCen) * RScl) + RWd), 255)
KeyG = clamp(0, (-abs((G - GCen) * GScl) + GWd), 255)
KeyB = clamp(0, (-abs((B - BCen) * BScl) + BWd), 255)
KeyA = min(KeyR, KeyG, KeyB)
```

The center, scale, and width parameters have the following meanings:

11573189

Center	Defines the color intensity at which the key is active, 0-255.
Scale	(255/(size of soft edge)). For hard edge keying, set scale to 255.
Width	(Size of half the key window including the soft edge)*scale. If width > 255, then keying is disabled for that channel.

In two-cycle mode, the keying operation must be specified in the second cycle (key alpha is not available as a combine operand). The combine mode G_CC_CHROMA_KEY2 is defined for this purpose.

The command

```
gsDPSetCombineKey(G_CK_KEY);
```

enables chroma keying.

The commands

```
gsDPSetKeyR(cR, sR, wR);
gsDPSetKeyGB(cG, sG, wG, cB, sB, wB);
```

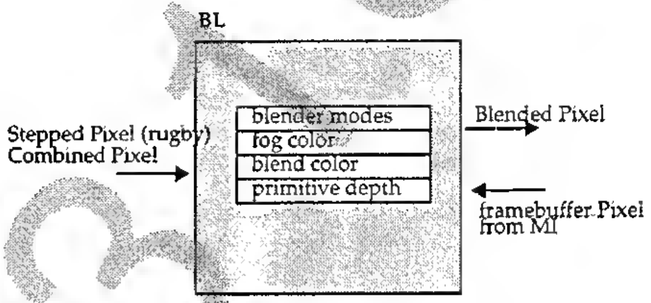
allow you to set the parameters for each channel.

1157389

BL: Blender

The BL takes the combined pixels and blends them against the framebuffer pixels. Transparency is accomplished by blending against the framebuffer color pixels. Polygon edge antialiasing is performed, in part, by the BL using conditional color blending based on depth range. The BL can also perform fog operations in two-cycle mode.

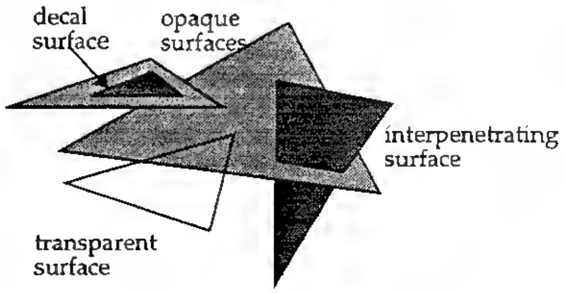
Figure 13-13 Blender State and Input/Output


BL
Stepped Pixel (rugby)
Combined Pixel
blender modes
fog color
blend color
primitive depth
Blended Pixel
framebuffer Pixel
from MI

Surface Types

The BL can perform different conditional color-blending and z-buffer updating. Therefore, it can handle semantically different surface and line types. Figure 13-14 illustrates these types.

Figure 13-14 Surface Types


decal
surface
opaque
surfaces
interpenetrating
surface
transparent
surface

11573189

Antialiasing Modes

The most important feature of the BL is its participation in antialiasing. Basically, the BL conditionally blends or writes pixels into the framebuffer based on depth range. Then the video display logic applies a spatial filter to account for surrounding background colors to produce antialiased silhouette edges.

The antialiasing scheme properly antialiases most pixels; only a small set of corner cases have errors and are negligible. This algorithm requires ordered rendering sorted by surface or line types. Here is the rendering order and surface/line types for z-buffer antialiasing mode:

- All opaque surfaces are rendered.
- All opaque decal surfaces are rendered.
- All opaque interpenetrating surfaces are rendered.
- All of the translucent surface and lines are rendered last. These can be rendered in any order. However, the proper depth order gives proper transparency.

Note: There is an additional optimization discussed later; if z-buffered surfaces in the scene are rendered in approximately front-to-backorder, the fill rate is improved because the z-buffer test is a read only (no write) for obscured pixels.

Besides the antialiased z-buffer rendering mode, the other three combinations also exist: antialiased/not z-buffered, z-buffered/not antialiased, not z-buffer/not antialiased.

Table 13-24 One-Cycle Mode gsDPSetRenderMode(mode1, mode2)

Parameter	Value
mode1	G_RM_FOG_SHADE_A G_RM_FOG_PRIM_A G_RM_PASS or one of the primitive rendering modes. e.g. G_RM_AA_ZB_OPA_SURF
mode2	e.g. G_RM_AA_ZB_OPA_SURF2

Note: Even if you are only in one-cycle mode, mode2 should be programmed. Mode2 value is always mode1 appended with "2".

Table 13-25 Two-Cycle Mode gsDPSetRenderMode(mode1, mode2)

Parameter	**Value**
mode1	G_RM_FOG_SHADE_A G_RM_FOG_PRIM_A G_RM_PASS
mode2	same as one cycle mode mode2 values

Note: When setting the cycle type to G_CYC_FILL or G_CYC_COPY, make sure to use the command *g*DPSetRenderMode(G_RM_NOOP, G_RM_NOOP2),* to guarantee that the blender is in a safe state.

BL Internal Color Registers

BL has two internal color registers, fog and blend color. These values are programmable and can be used for geometry with fog or constant transparency.

Table 13-26gsDPSetFogColor(r, g, b, a) gsDPSetBlendColor(r, g, b, a)

Parameter	**Value**
r, g, b, a	color

Alpha Compare

BL can compare the incoming pixel alpha with a programmable alpha source to conditionally update the framebuffer. This has traditionally allowed nice tree-outlined billboards and other complex, outlined, billboard objects.

1153789

11513189

Besides thresholding against a value, the BL can also compare against a dithered value to give randomized particle effect:

Table 13-27 gsDPSetAlphaCompare(mode)

Parameter	Value
mode	G_AC_NONE G_AC_THRESHOLD G_AC_DITHER

Note: When using mode G_AC_THRESHOLD, alpha is thresholded against blend color alpha.

Note: Another way to do billboard cutouts which often provides better antialiasing is ti turn Alpha Compare off (G_AC_NONE) and instead use one of the TEX_EDGE render modes, such as G_RM_AA_ZB_TEX_EDGE.

Using Fog

The blender performs the fog operation. Fog is described fully in "Vertex Fog State" on page 169. Fog is performed by the RSP and the RDP in cooperation. The RSP takes the z value and places it in the alpha channel of each pixel. The RDP then uses this alpha channel to blend the color from the color combiner with the fog color. The larger the Z value (the farther the pixel is from the viewers eye) the closerthe pixel's color gets to the fog color. The RSP part of this operation is enabled with the gSPSetGeometryMode:

```
gsSPSetGeometryMode(G_FOG),
```

and can be adjusted with gsSPFogPosition:

```
gsSPFogPosition(FOG_MIN, FOG_MAX),
```

The RDP part of fogging is enabled by telling the blender how to use Alpha. Fog can be used in one cycle mode for non-antialiased opaque surfaces only:

```
/* 1cycle mode */
gsDPSetCycleType(G_CYC_1CYCLE),
/* blend fog in ZB mode (non-AA OPA_SURF modes only) */
gsDPSetRenderMode(G_RM_FOG_SHADE_A,G_RM_ZB_OPA_SURF2),
/* set the fog color */
gsDPSetFogColor(RED,GREEN,BLUE, ALPHA),
/* setup the RSP */
gsSPFogPosition(FOG_MIN, FOG_MAX)
gsSPSetGeometryMode(G_FOG),
```

It can be used for other surface types (or with antialiasing) in 2 cycle mode:

```
/* 2 cycle mode */
gsDPSetCycleType(G_CYC_2CYCLE),
/* blend fog. Use any standard render mode for cycle 2 */
gsDPSetRenderMode(G_RM_FOG_SHADE_A,G_RM_AA_ZB_OPA_SURF2),
/* set the fog color */
gsDPSetFogColor(RED,GREEN,BLUE, ALPHA),
/* setup the RSP */
gsSPFogPosition(FOG_MIN, FOG_MAX)
gsSPSetGeometryMode(G_FOG),
```

As an alternative to G_RM_FOG_SHADE_A (for the first cycle of gsDPSetRenderMode) you can use G_RM_FOG_PRIM_A which will use the alpha value in PrimColor to set the fog value. If you use this mode, then the RSP's part of fog is unneccessary and the gsSPFogPosition and gsSPSetGeometryMode macros are not neccessary. Instead set the fog value per primitive with the gsDPSetPrimColor macro:

```
gsDPSetPrimColor(0,0,0,0,0, FOG_VALUE),
```

where the FOG_VALUE is 0 for no fog and 0xff for full-fog.

Note that objects with FOG can still be transparent. The alpha value used to modulate fog comes from the triangle renderer. The alpha value that comes from the color combiner is independant of that renderer fog alpha. For example the color combiner can be set to use the alpha value from a texture map, and fog will still work with the alpha value from the renderer. You cannot, however, use vertex alpha with fog. The per alpha supplied in the vertices will be ignored and if the color combiner selects a SHADE alpha, it will get the fog alpha value instead (not what was intended).

115789

Depth Source

The depth value used in the depth buffer compare is generally taken from the Z value of the pixel, determined by interpolating the z values at the 3 vertices of the triangle containing the pixel. However it is sometimes desireable to set the Z value which will be used for an entire primitive. This is actually neccessary when rendering Z-buffered rectangles (gDPFillRect and gSPTextureRect) since these primitives do not have a Z value associated with them. To use a single Z value for an entire primitive the Z value is placed in the PrimDepth register and the Z source Select is set to get Z from the PrimDepth register:

```
gsDPSetDepthSource(G_ZS_PRIM),
gsDPSetPrimDepth(z, dz),
```

The value to use for z is the screen Z position of the object you are rendering. This is a value ranging from 0x0000 to 0x7fff, where 0x0000 usually corresponds to the near clipping plane and 0x7fff usually corresponds to the far clipping plane. To synchronize Z for PrimDepth with a Z for a triangle it is important to understand how the triangle's Z gets computed. The modeling coordinate vertex is multiplied by the modelview and projection matrices resulting in a 4 componant homogeneous coordinate (x,y,z,w). The screen Z value is computed by the RSP as

```
screenZ = 32*((z/w)*Viewport.vscale[2] + Viewport.vtrans[2])
```

Note: Viewport.vscale and Viewport.vtrans[2] are usually both G_MAXZ/2 = 0x1ff, which makes the formula: screenZ=(z/w)*0x3fe0 + 0x3fe0. Since (z/w) ranges from -1.0 to +1.0 the result will range from 0x0 to 0x7fc0.

Note: For microcode progrmmers: The 32* part of this equation is done in the setup microcode. The other parts of this equation are done in the vertex processing microcode.

So if you want to position a rectangle at a specific modeling coordinate position, run the modeling ccordinate of the position through the modelview and projection matrix, and then comput its screenZ value based upon the formula above. This is the value to use for z in the gsDPSetPrimDepth command.

The dz value should be set to 0. This value is used for antialiasing and objects drawn in decal render mode and must always be a power of 2 (0, 1, 2, 4, 8, ... 0x4000). If you are using decal mode and part of the decalled object is not being rendered correctly, try setting this to powers of 2. Otherwise use 0.

11573189

MI: Memory Interface

Memory Interface (MI) simply interfaces to the framebuffer memory. It has programmable color and z-buffer pointers, a 32-bit fill color value used in the FILL cycle type (see Chapter 13, "Fill Mode"), and an enable for color dither.

Figure 13-15 Memory Interface State and Input/Output

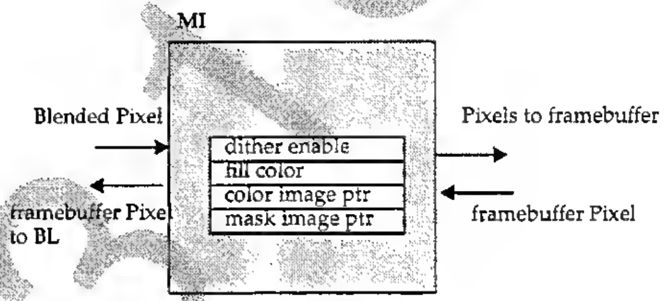
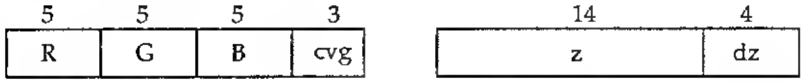

MI
Blended Pixel
framebuffer Pixel to BL
dither enable
fill color
color image ptr
mask image ptr
Pixels to framebuffer
framebuffer Pixel

Image Location and Format

The framebuffer is row-ordered, starting at the upper left. The color and z-buffer image pointers must be 64-byte aligned. The DRAM has dual banks, one on each 1 MB. By keeping the color and z-buffers on different banks, you can improve the DRAM access latency when the RDP is seeking DRAM bandwidth for rendering.

The Nintendo 64 system actually uses 9-bit DRAMs rather than 8-bit DRAMs, to gain two extra bits per color or z pixel. The color and z format are illustrated in Figure 13-16.

Figure 13-16 Color and Z Image Pixel Format


5
5
5
3
R
G
B
cvg
14
4
z
dz

Fill Color

The MI has a 32-bit fill color register that is used in FILL cycle type. Fill color is typically programmed to a constant value to fill background color and z-buffers. Since two framebuffer pixels are 18x2=36 bits, while fill color register is 32 bits, a few of the bits are replicated. See Figure 13-17 for an illustration of how it works.

Figure 13-17 Fill Color Register LSB Replication

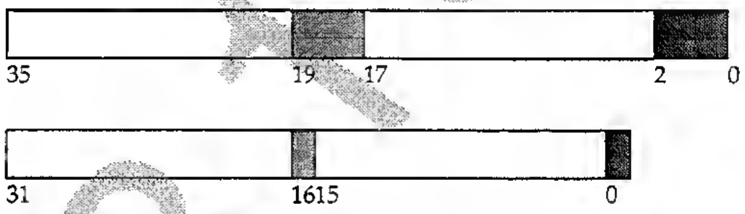


Table 13-28 gsSetFillColor(data32bits) NEED READABLE TITLE FOR THIS!

Parameter	Value
data32bits	2 different macros, one each for color and z. each generate 16 bits. so do x << 16 | x to get 32 bits GPACK_RGBA5551(r, g, b, a), a=1 is full coverage. (Typical) GPACK_ZDZ(z, dz), z=G_MAXFBZ, dz=0. (Typical)

Dithering

The RDP pipeline keeps full, 8-bit per RGB component precision throughout. Dithering can be enabled or disabled to write to the 5-bit per RGB component dram framebuffer format. Dithering is recommended since it can significantly reduce Mach banding effect.

1157389

1157318 9

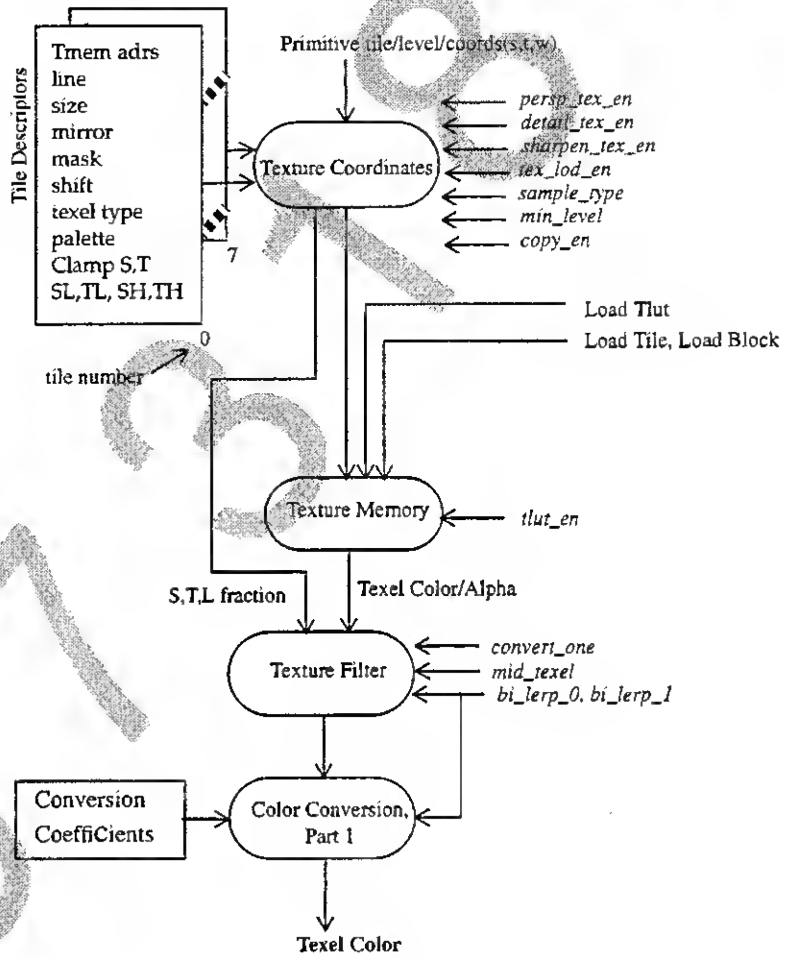
Chapter 14

Texture Mapping

1157389

Texture mapping, or texturing, is the process of applying an image to a polygonal surface. There are many graphics books that discuss this topic; this guide assumes that you are familiar with the basic principles of texture mapping. This chapter explains the functionality of texture mapping as implemented in the Reality Display Processor (RDP).

Figure 14-1 Texture Unit Block Diagram


Tile Descriptors
Tmem adrs
line
size
mirror
mask
shift
texel type
palette
Clamp S,T
SL,TL, SH,TH
7
0
tile number
Primitive tile/level/coords(s,t,w)
Texture Coordinates
persp_tex_en
detail_tex_en
sharpen_tex_en
tex_lod_en
sample_type
min_level
copy_en
Load Tlut
Load Tile, Load Block
Texture Memory
tlut_en
S,T,L fraction
Texel Color/Alpha
Texture Filter
convert_one
mid_texel
bi_lerp_0, bi_lerp_1
Conversion CoeffiCients
Color Conversion, Part 1
Texel Color

The RDP contains an on-chip texture memory called Tmem, which buffers all source image data used for texturing. Tmem contains up to eight tiles (a tile is a rectangular region of an image). A tile is loaded into Tmem using the *LoadTile*, *LoadBlock*, or *LoadTlut* commands, and described using the *SetTile* and *SetTileSize* commands. If the image is too large to fit entirely in Tmem, primitives must be subdivided in object space based on their texture coordinate values so that each primitive references a tile that fits in Tmem.

Texture coordinates (S,T) for each pixel are input to the texture coordinate unit and can be perspective corrected. Perspective correction is typically enabled for 3D geometry and disabled for 2D sprites (*tex_rect commands*). During this time, the texture coordinate unit calculates which tile descriptor to use for this primitive. The texture image coordinates are converted to tile-relative coordinates and wrapped, mirrored, and clamped. These tile coordinates are then used to generate an offset into Tmem. The texture unit can address 2x2 regions of texels in one or two cycle mode, or 4x1 regions in copy mode. Copy mode is typically used for blits (block copy of texels) with a 1:1 texel pixel relationship. In one or two cycle mode, filter or point-sample can also be selected. Typically, filter will result in a smoother image with less aliasing. The texture unit also generates S,T and L-fraction values that are used to bi-linearly or tri-linearly interpolate the texels.

The texture unit supports ten different combinations of texel size and format:

- 4-bit intensity (I)
- 4-bit intensity w/alpha (I/A) (3/1)
- 4-bit color index (CI)
- 8-bit I
- 8-bit IA (4/4)
- 8-bit CI
- 16-bit red, green, blue, alpha (RGBA) (5/5/5/1)
- 16-bit IA (8/8)
- 16-bit YUV (Luminance, Blue-Y, Red-Y)
- 32-bit RGBA (8/8/8/8)

Significant memory savings can result from the smaller color-index textures or intensity textures over the more expensive 16-bit RGBA. It is a good idea to experiment with the different texel sizes. One can actually do 2-color textures using the intensity types. Also, the intensity-only textures place the texel value on the alpha channel as well where it can be used for blending or ignored.

1157389

Graphics Binary Interface for Texture

The graphics binary interface (GBI) is a set of macros that create 64-bit commands that are read and parsed by the RSP microcode. Some of these commands cause actions or state changes in the RSP. Others are simply passed through the RSP to the RDP. Below is a list of GBI commands that control texture. See the corresponding reference (man) page for more details.

Primitive Commands

- g*SPTexture
- g*SPTextureRectangle*

Tile Related Commands

- g*DPSetTile
- g*DPSetTileSize

Load Commands

- g*DPLoadTile*
- g*DPLoadTextureBlock*
- g*DPLoadTLUT*
- gDPSetTextureImage

Sync Commands

- g*DPLoadSync
- g*DPTileSync

Mode Commands

- g*DPSetTextureLUT
- g*DPSetTexturePersp

11573189

- g*DPSetTextureDetail
- g*DPSetTextureLOD
- g*DPSetTextureFilter
- g*DPSetTextureConvert

115731189

Example Display List

The following display list fragment uses GBI display list commands to render an object using a 16-bit RGBA texture map. The texture is loaded into Tmem using the *LoadBlock* command. The texture coordinates are perspective corrected. Note that the texture is allowed to wrap on 32-texel boundaries in the s and t directions. The texture filter bilinearly interpolates the 2x2 texels output by the texture unit. Finally, the resulting texture color is multiplied with the object's shade color in the Color Combiner for each pixel of the object.

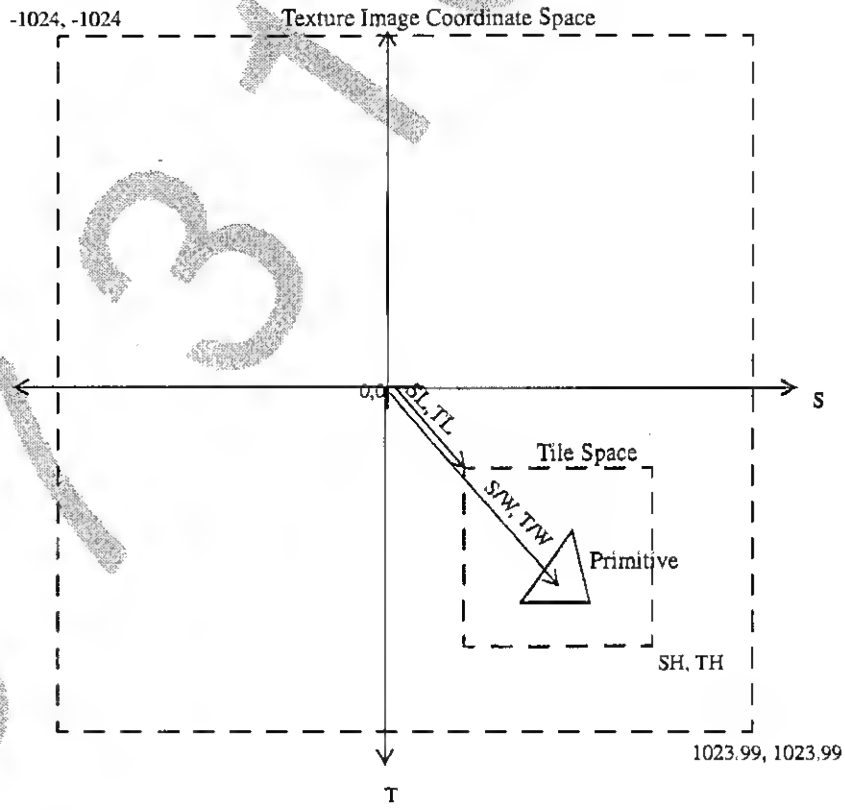
```
/* Enable textured poly generation in RSP */
gSPTexture(glistp++, 0x8000, 0x8000, G_TX_RENDERTILE, G_ON);
gDPSetTextureFilter(glistp++, G_TF_BILERP);
gDPSetTexturePersp(glistp++, G_TP_PERSP);
gDPSetCombineMode(glistp++,
G_CC_MODULATERGB, G_CC_MODULATERGB);
/* Load Texture Block */
gDPLoadTextureBlock(glistp++, RGBA16dana, G_IM_FMT_RGBA,
G_IM_SIZ_16b, 32, 32, 0, G_TX_WRAP, G_TX_WRAP, 5, 5,
G_TX_NOLOD, G_TX_NOLOD);
/* render model display list */
gSPDisplayList(glistp++, model);
```

11573189

Texture Image Space

Texture coordinates are defined for textured primitives in Texture Image Space. This space has a range of +/- 1K texel. Tiles are smaller rectangular regions of a texture that fit into the on-chip texture memory of the RCP (Tmem).

Figure 14-2 Image Space and Tile Space


-1024, -1024
Texture Image Coordinate Space
0,0
SL, TL
Tile Space
S/W, T/W
Primitive
SH, TH
S
T
1023.99, 1023.99

Tiles are defined in Texture Image Space using SL, TL and SH, TH coordinates, as shown in Figure 14-2. Tile coordinates must lie in the positive S,T quadrant of Texture Image Space. However texture coordinates of the primitive can lie in any of the four quadrants of image space. In other words, primitives can have negative texture coordinates which can be useful when wrapping a texture on a very large primitive. Tiles can be up to 1024 columns wide and up to 256 rows tall. Tiles do not have to be sized to a power of 2 (wrapping and mirroring, however, happen on power-of-2 boundaries).

The texture coordinates of the primitive (in Texture Image Space) are converted into Tile Space by subtracting the SL,TL from the (possibly perspective-corrected) texture coordinates of the pixel. This indirection allows arbitrary placement of the tile with respect to the primitive. This implies that the texture coordinates can be defined once in the database; and that the texture can be translated (or slid) with respect to the primitive by simply manipulating the SL,TL values using the *SetTileSize* RDP command.

1157319

Tile Attributes

The RDP has a small on-chip memory for buffering up to eight tile descriptors at a time. A tile descriptor contains all the information for a texture tile including format; size; line; Tmem address; palette; mirror enable S, T; mask S, T; shift S, T; SL, TL, SH, TH, and clamp S, T.

Format

Format of texels in texture tile.

Table 14-1 Tile Format Encodings

Format Value	Format
0	RGBA
1	YUV
2	CI
3	IA
4	I

Size

Size of texels in texture tile.

Table 14-2

Size Value	Size of texel in bits
0	4
1	8
2	16
3	32

11573189

Line

Number of 64-bit words in one row of the tile. Dependent on tile row width as well as texel type/size. When tiles are loaded using the *LoadTile* command, the rows are padded to 64-bit boundaries. When *LoadBlock* is used to load a texture, it is assumed that the rows have already been padded. Line can also be used to control the stride through TMEM. By controlling Line, smaller tiles can be pieced together into one larger continuous tile.

Tmem Address

Tile offset (0-511) in Tmem (64-bit) words.

Palette

Palette number (0-15) of 4-bit Color Index (CI) textures. An 8-bit index into the high half of Tmem is formed by placing the palette number in the 4 MSBs and the 4-bit texel value in the 4 LSBs. The color in Tmem at this index becomes the color of the pixel. Therefore, for a 4-bit CI texture, you may select one of 16 palettes with each palette having up to 16 entries. Palettes can be loaded into Tmem using the *LoadTLUT* command or, optionally, the *LoadBlock* command.

Mirror Enable S,T

Enables mirroring of texture coordinates. When the bit indicated by the (Mask Value + 1) is 0 the coordinates are unchanged. When this bit is 1, however, the coordinates are inverted. Useful for symmetric patterns like trees, faces, etc. For example, a mask of 2 with mirror enabled would yield the following texture coordinates:

```
0,1,2,3,4,5,6,7,...  Input coordinate
0,1,2,3,3,2,1,0,...  Mirrored Coordinate
```

Mask S,T

Number of bits of tile coordinate to let through. For example, a mask of 1 indicates one bit of the texture coordinate should come through the mask, giving a pattern of 0,1,0,1... As another example, a mask value of 5 indicates that the texture should wrap every 32 texels, i.e., the lower 5 bits are passed through the mask. A mask value of 0 forces clamping the texture coordinates to be between (SL,TL),(SH,TH) inclusive. The mask value + 1 indicates the bit position that is looked at for mirroring. See discussion in Mirror Enable, above.

1157319

Shift S,T

Shift texture coordinates after perspective divide. Used in MIP maps and possibly for precision reasons (see the discussion of Detail texture later in this document). Also useful for combining two differently scaled textures.

Table 14-3 Shift Encoding

Shift Value	**Shift**
0	no shift
1	>> 1
2	>> 2
3	>> 3
4	>> 4
5	>> 5
6	>> 6
7	>> 7
8	>> 8
9	>> 9
10	>> 10
11	<< 5

Table 14-3 Shift Encoding

Shift Value	Shift
12	<< 4
13	<< 3
14	<< 2
15	<< 1

SL,TL

When rendering, the starting texel column, row of tile in texture image space, 10.2 fixed point. Can be used to slide texture w.r.t. the primitive. When loading, the starting texel column, row within the DRAM texture image.

SH,TH

When rendering, the ending texel column, row of tile in texture image space, 10.2 fixed point. Used for clamping only. When loading, the ending texel column, row within the DRAM texture image.

Clamp S,T

Enable clamp during wrap or mirror. When not masking, Clamp S,T is ignored and clamping is implicitly enabled. This bit allows clamping the texture coordinates when the mask is non-zero. Useful when you want to mirror and then clamp like an airplane wing insignia. The border of the insignia would have an alpha of 0. For example, SH = 11, mask = 2, mirror = 1, clamp = 1:

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,...  Input Coordinate
0,1,2,3,3,2,1,0,0,1, 2, 3, 3, 3, 3, 3,...  Mirrored/Clamped
Coordinates
```

11513189

Tile Descriptor Loading

Tile descriptors must be loaded using the RDP command SetTile. This command loads the *format, size, line, Tmem address, palette, clamp, mirror, mask,* and *shift* parameters for the tile number specified. The *SL, TL, SH,* and *TH* parameters are set by the RDP commands *SetTileSize, LoadTile, LoadBlock,* and *LoadTLUT*.

One important point to keep in mind is that tile descriptors are used both when loading textures and when rendering textures. In particular, when loading a texture, the texture coordinate unit uses the Tmem address, line, format, and size information from the tile specified in the *LoadTile/Block/TLUT* command. Therefore, this information must be loaded into the tile descriptor prior to executing the *LoadTile/Block/TLUT* command. Also, the *LoadTile/Block/TLUT* command automatically writes the *SL,TL,SH,TH* information into the tile descriptor. In the case of a *LoadTile* command, this is probably the information you wanted. In the case of a *LoadBlock* or *LoadTLUT* command, however, this information must be overwritten with a *SetTileSize* command after the texture load.

The GBI commands for loading tile descriptors directly are:

- g*DPSetTile
- g*DPSetTileSize

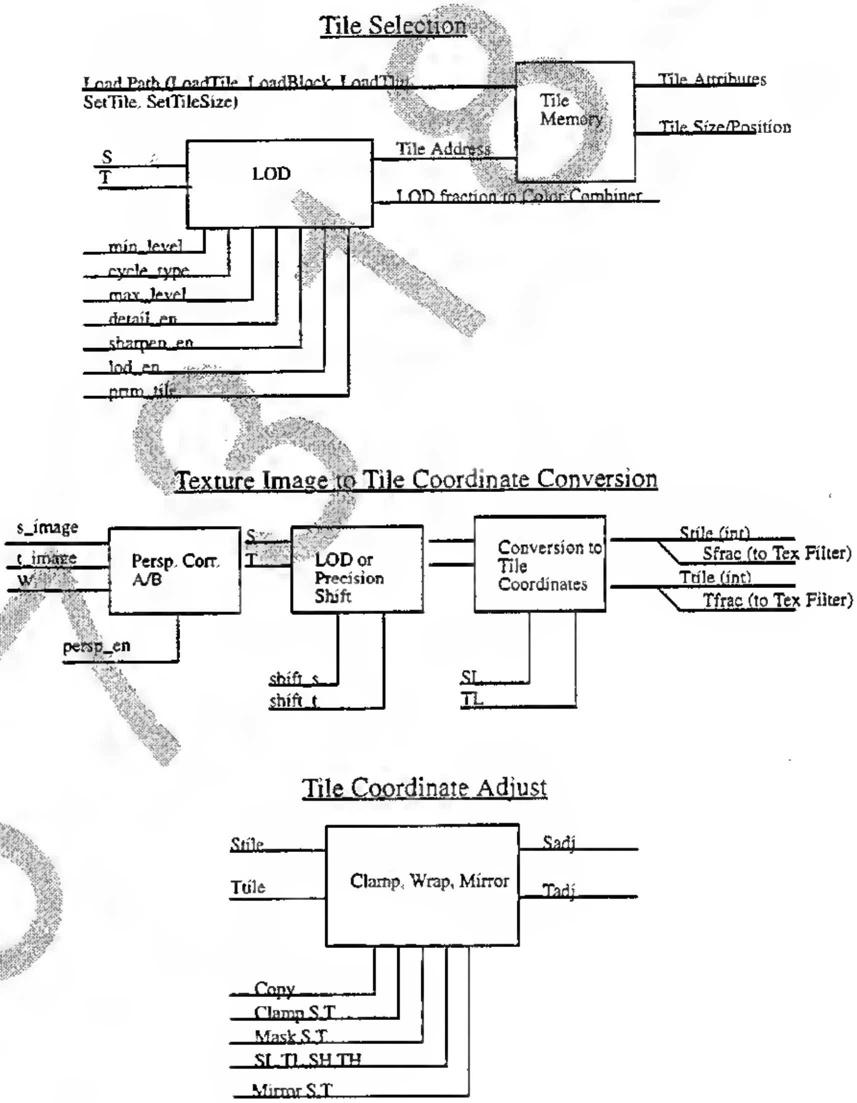
The GBI commands that effect tile descriptors are:

- g*DPLoadTile*
- g*DPLoadTextureBlock*
- g*DPLoadTLUT*

Note: The load commands above use a double buffered tile system for loading/rendering. When loading, the tile G_TX_LOADTILE is used, and when rendering the tile G_TX_RENDERTILE is used. This simple scheme avoids having to insert *TileSyncs* between loading and rendering. However, if you need to use more than one tile for some reason, make sure that you use the *g*DPSetTile* and *g*DPSetTileSize* to set the tile descriptors properly.

Texture Pipeline

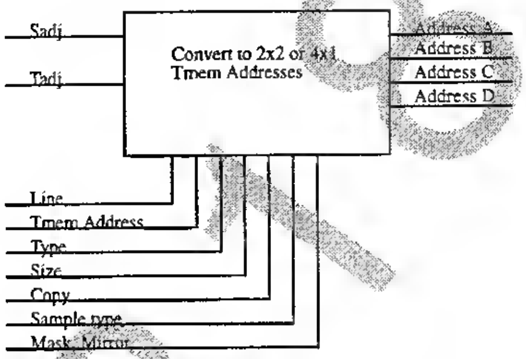
Figure 14-3 Texture Pipeline


Tile Selection
Load Path (LoadTile, LoadBlock, LoadTlut, SetTile, SetTileSize)
Tile Memory
Tile Attributes
Tile Size/Position
S
T
LOD
Tile Address
LOD fraction to Color Combiner
min_level
cycle_type
max_level
detail_en
sharpen_en
lod_en
prim_tile
Texture Image to Tile Coordinate Conversion
s_image
t_image
W
Persp. Corr. A/B
persp_en
S
T
LOD or Precision Shift
shift_s
shift_t
Conversion to Tile Coordinates
SL
TL
Stile (int)
Sfrac (to Tex Filter)
Ttile (int)
Tfrac (to Tex Filter)
Tile Coordinate Adjust
Stile
Ttile
Clamp, Wrap, Mirror
Sadj
Tadj
Copy
Clamp S,T
Mask S,T
SL,TL,SH,TH
Mirror S,T

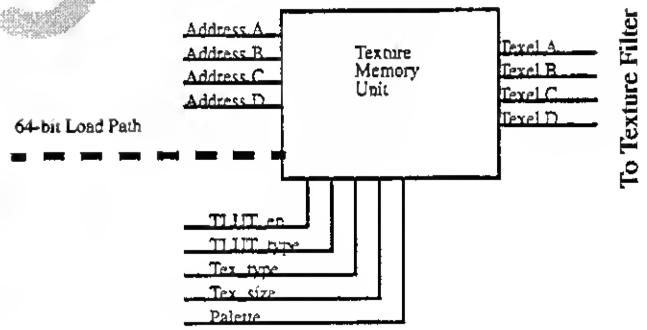
11573199

Figure 14-4 Texture Pipeline, contd.

Tile Coordinate To Address Conversion


Sadj
Tadj
Convert to 2x2 or 4x1 Tmem Addresses
Address A
Address B
Address C
Address D
Line
Tmem Address
Type
Size
Copy
Sample type
Mask Mirror

Texture Memory


Address A
Address B
Address C
Address D
64-bit Load Path
Texture Memory Unit
Texel A
Texel B
Texel C
Texel D
To Texture Filter
TLUT_en
TLUT_type
Tex_type
Tex_size
Palette

1157319

115136809

Tile Selection

Functionality

Tile descriptors are used both when loading a texture and when rendering a texture. This section discusses the selection of tiles when rendering. The use of tiles descriptors when loading textures in discussed in the **Loading Textures** section.

There are basically two ways to index into tile memory: explicitly via a user-defined tile number, or indirectly using a combination of the user-defined tile number and the level of detail (LOD) of the pixel.

In two-cycle mode, it is possible to access different tile descriptors in each cycle. The computation of tile indices for each cycle depends on several mode bits and is described in the following sections.

LOD Disabled

With **LOD disabled**, the user specifies the texture tile for a primitive directly using the *gSPTexture* command. This tile number is inserted by microcode into the header for each subsequent primitive and is referred to as the *primitive tile number.* 2-cycle non-LOD mode can be useful for combining two arbitrary textures (morphing, etc.) The calculation of the tile descriptor index is straight forward when LOD is disabled:

Table 14-4 Tile Descriptor Index Generation with LOD Disabled

Cycle	Tile Index
0	primitive tile
1	primitive tile + 1

1157389

LOD Enabled

The lod_en mode bit in *SetOtherModes* determines if tile indices are determined using Level of Detail (LOD) or from the primitive command directly.

With LOD enabled, the tile index is a function of the Level of Detail (LOD) of the primitive. LOD is computed as a function of the difference between perspective corrected texture coordinates of adjacent pixels to indicate the magnification/minification of the texture in screen space (texel/pixel ratio). The LOD module also calculates an LOD fraction for third axis interpolation between MIP maps. The combination of LOD-derived tile coordinates and fraction, a particular tile descriptor arrangement, and tri-linear filtering allows the implementation of MIP maps. Notice that MIP mapping is a specialized use of the general texture hardware. Other types of mappings are possible. The LOD calculation makes the following features (and maybe more) possible:

- trilinear MIP mapping
- sharpened texture
- detail texture

The LOD calculation depends on the following inputs:

- **LOD:** level of detail@pixel (texels/pixel), derived per pixel
- **min_level (0.5):** minimum LOD fraction clamp for sharpen or detail modes, from the *SetPrimColor* RDP command
- **max_level (0-7):** number of MIP maps minus one, from the primitive via the *gSPTexture* command.
- **detail_en:** enable for detailed texture, from *SetOtherModes* RDP command
- **sharp_en:** enable sharpen mode, from *SetOtherModes* RDP command
- **prim_tile (0-7):** primitive tile number, from the primitive via the *gSPTexture* command.
- **lod_en:** enable for LOD calculation, from *SetOtherModes* RDP command

The LOD calculation produces the following outputs:

- **l_frac (s,0.8):** LOD fraction for 3rd axis interpolation
- **l_tile (0-7):** tile descriptor index into tile memory

The LOD per pixel is clamped to *min_level*. The LOD tile index is then calculated using the equation:

l_tile = log2((int)lod_clamp)

So, for example, an LOD of 7.5 would be converted to an *l_tile* of 2. This index is clamped to *max_level* and then added to the *prim_tile*. For example, the tile arrangement for a MIP map with a *prim_tile* = 2 and *max_level* = 3 would be arranged as shown in Table 14-5.

Table 14-5 Example of Tile Address and LOD Index Relationship

Tile Address	LOD Index
0	-
1	-
2	0
3	1
4	2
5	3
6	-
7	-

The *l_frac* is derived by dividing the clamped LOD by 2 $^{l\_tile}$. For example, an LOD of 7.5 would yield an *l_frac* of 0.875. The *l_frac* is modified depending on the mode bits *detail_en* and *sharp_en*. Note that the detail and sharpen modes discussed below are exclusive. If enabled simultaneously, special effects may result. If neither *detail_en* or *sharp_en* is true, then the *l_frac* is passed to the color combiner unmolested.

Sharpen and detail mode change the behavior of the tile index calculation when magnifying. The texture is magnified when you get so close to the

11573189

primitive that one texel is being applied to many pixels, even using the highest resolution texture in the MIP map.

Table 14-6 Generation of Tile Descriptor Index With LOD Enabled and Magnifying

Cycle	Detail	Sharpen	!Detail & !Sharpen
0	prim_tile + l_tile	prim_tile + l_tile	prim_tile + l_tile
1	prim_tile + l_tile + 1	prim_tile + l_tile + 1	prim_tile + l_tile

Table 14-7 Generation of Tile Descriptor Index With LOD Enabled and Not Magnifying

Cycle	Detail	Sharpen	!Detail & !Sharpen
0	prim_tile + l_tile + 1	prim_tile + l_tile	prim_tile + l_tile
1	prim_tile + l_tile + 2	prim_tile + l_tile + 1	prim_tile + l_tile + 1

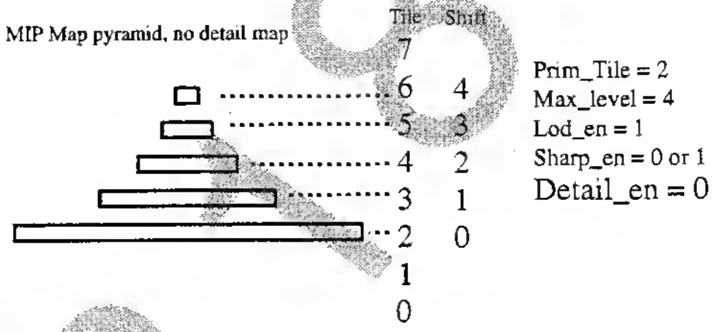
Also note that *l_tile* is clamped to *max_level* when at the coarsest level of detail.

11573189

MIP Mapping

An example of the tile arrangement for a MIP map is shown in Figure 14-5.

Figure 14-5 MIP Map Tile Descriptors



To implement trilinear MIP mapping, the RDP must be in two-cycle mode. A tile is referenced in each of the cycles and linearly interpolated using the *l_frac* in the color combiner.

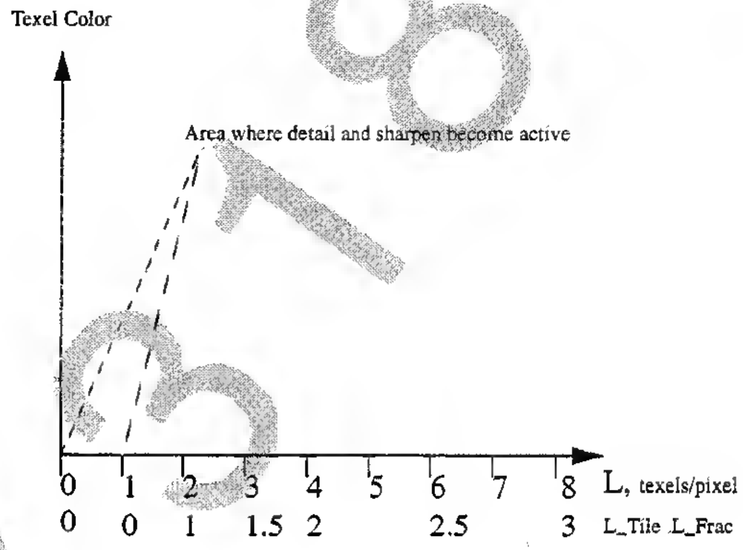
For more control of interpolation between two texture tiles a register *prim_frac (0.8)* is provided that can be used as an input to the color combiner. prim_frac is set by the *SetPrimColor* command.

Care should be taken in the off-line generation of the MIP maps. Depending on the filter used for generating the levels, the different levels can end up unaligned if not careful. For example, if using a simple box filter for generating the coarser levels, an offset of 0.5 should be added to the SL and TL of each level to insure that they align when laid on top of one another. Whether these or other offsets are necessary depends on the filter used. Typically higher order filters will result in higher quality MIP maps.

Another word of caution. In computer graphics, extremely high frequency textures are a bad thing. Going from black to white in one texel being the highest frequency. High frequency maps are more likely to alias (flicker) when edge on or far away. So when generating map data use common sense and possibly lower frequency texture data to avoid these problems.

Magnification

Figure 14-6 Magnification Interval Relative to LOD



Detail Texture

Even with trilinear MIP mapping, textures can look blurry under magnification (that is, when 0.0 < LOD <= 1.0). One way of avoiding this is to use very large textures that contain high-frequency detail. But this would be expensive in Tmem.

Detail mode comes into play in magnification. The finest level of the base texture is combined with a (usually small) detail texture in such a way as to repeat the detail-texture over the base texture several times. A base-texel would, upon magnification, appear to contain four or more detail texels blended with the base-texel color, thus providing high-frequency information without having to sacrifice large amounts of Tmem. This can be used very effectively; for example, to provide motion cues when close to the terrain.

11573189

Detail texture mode is most effective in situations where the high-frequency information and overall hue are relatively consistent throughout the texture. To convert a high-resolution image into a low-resolution image (for the base texture) and a detail texture, follow this procedure:

12. Make the low-res image by filtering the high-res image to the desired size. This will become the base level.
13. Any nxn sub-tile of the high-res image can be used as a detail-texture. This sub-tile should preferably be modified to match across s and t borders so that when it is repeated on the base-texture, the seams are not visible. Detail textures can have a different texel type than the base-texture (subject to Tmem restrictions). Often, it is sufficient to use a 4-bit or 8-bit intensity detail-texture

A very effective and efficient implementation of detail texture involves use of the base texture itself as the detail texture but at a different resolution. This works well for objects and terrains with a 'fractal character' where different resolutions of the object look similar. In such cases it might be appropriate to set the *min_level* parameter to 0 to allow the detail texture to completely replace the base texture at high magnifications.

Since the detail texture is combined with the base texture, a color shift may result. This can be avoided by choosing the detail texture color scheme to match the base texture colors so that this effect is minimized. The *min_level* parameter can also be used to keep the detail texture from completely replacing the base texture by setting it to a value greater than 0. This will cause a certain minimum amount of the base texture to always be blended in with the detail texture thus minimizing the color shift.

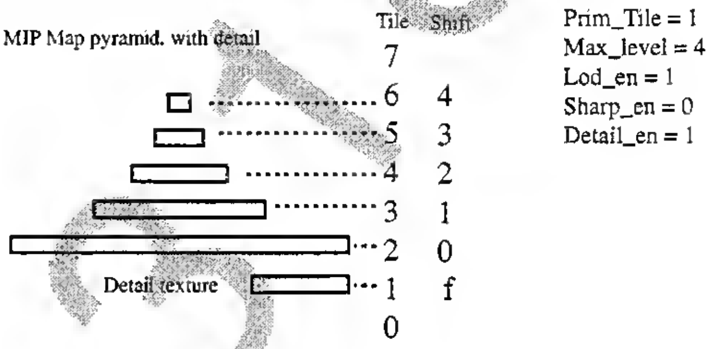
The *shift* field of the tile pointing to the detail texture is used to shift the incoming s and t coordinates before indexing into the map. This shift then determines the base-texel to detail-texel ratio.

For example, if the detail tile's *shift* was set to shift left by 1 (the shift of the finest level of the base texture being 0, of course), each base-texel, upon magnification would display 4 detail-texels blended with the base-texel color. A shift left of 2 would result in 16 detail-texels per base- texel and so on. Larger shifts result in more aliasing in the detail-texture since the interpolation occurs between widely different magnifications.

Keep in mind that the *shift* values compromise between the base-texel to detail-texel ratio and the effectiveness of the bilerp operation on the detail texture. This is because the number of fractional bits in the s and t coordinates (s10.5) is limited to 5 bits. Hence, a shift left of 3 bits will leave only 2 bits of fraction within each texel to do the bilerp.

Detail textures must always be pointed to using PRIM_TILE.

Figure 14-7 MIP Map With Detail Texture Tile Descriptors



If *detail_en* is true and the LOD is less than 1.0, indicating that the LOD is below the finest MIP map level, the fraction is a table lookup of the *l_frac*. Currently, the table lookup is simply identity, so the fraction is not modified in detail mode. In order to always to have a portion of the base-texture visible, *l_frac* is clamped to be greater than *min_level*. *Min_level* should be determined by experimentation. This fraction can then be used to interpolate between the detail-texture (pointed to by *prim_tile*) and the base-texture (pointed to by *prim_tile*+1). Filtering within the detail-texture can be controlled as usual by using the *setOtherModes* bits to be POINT or BILERP.

Sharpen Mode

Sharpen mode is used in a situation similar to that of detail texture. The advantage of sharpen over detail is that sharpen is essentially free. It doesn't require an additional detail map. Instead it extrapolates using the two finest MIP map levels. An image with high contrast edges has been magnified to the point where the edge details are becoming blurry. Sharpen mode increases the apparent sharpness of the texture edge by inverting the *l_frac* *(extrapolating)* as shown in Figure 14-8, "Sharpen Extrapolation," on page 238.

Bilinear Filtering and Point Sampling

The DP hardware treats texture coordinates differently based on whether the DP is in point sample mode or bilerp mode. In point sample mode texels can be thought of as 1 x 1 squares with the sample point at the top left hand corner of the texel (where the 's' and 't' coordinate axes run left to right and top to bottom respectively. This means that to map a modeler's floating point texture coordinate output (u,v) into the DP fixed point texture coordinates (s,t) for say a 32x32 sized texture (s ranges from 0 - 31 and t ranges from 0 - 31), the mapping

```
s = u*32;
t = v*32;
```

would work consistently and would map the full 32x32 texture onto a polygon with (u,v) coordinates in the range [0.0 - 1.0]. This is because the above mapping would result in u range of [0.0-1.0] to be mapped to an s range of [0-32] which would cover the region from the left edge of the texel 0 to the right edge of texel 31.

On the other hand, in Bilerp mode the DP treats a texel as a 1 x 1 square with the sample point at the center and the above mapping would cover the region from the middle of texel 0 to the middle of texel 32 which goes beyond the extent of the texture.

The mapping

```
s = u*32 - 1;
t = v*32 - 1;
```

doesn't work either since it maps a (u,v) range of 0.0 - 1.0 to an (s,t) range of 0.0 - 31.0 which would cover a region from the middle of texel 0 to the middle of texel 31 which cause both texel 0 and texel 31 to be half displayed.

The mapping that would make the textured primitive match exactly to the artist's rendition of the texture in Bilerp mode would be:

```
s = u*m - 0.5;
t = v*n - 0.5;
```

since this would map a (u,v) range of [0.0-1.0] to an (s,t) range of [-0.5 - 31.5] which would cover the region starting on the left edge of texel 0 to the right edge of texel 31. However the bilerp filter requires two texels to bilerp between and in the s,t ranges [-0.5 - 0.0] and [31.0 - 31.5] there is only one texel available. This can be solved by turning on clamping in the DP and setting SL,TL to 0,0 and SH,TH to 31,31. This will cause the bilerp filter to select texel 0 for both texels to bilerp between in the range [-0.5 - 0.0] and texel 31 for range [31.0 - 31.5]. This paradigm can be extended for wrapping textures by clamping only at the border coordinates of the primitive. For example a primitive with u,v in the range [0.0-4.0] in wrap mode would repeat the texture 4 times. For the border texels to be displayed in full the s,t range would have to be [-0.5 - 127.5] (according to the above mapping) and the clamp parameters SL,TL and SH,TH would be set to 0,0 and 127,127 respectively. (Note that SL and TL is subtracted from the incoming texture coordinates and is also used as the lower clamp value in clamp mode).

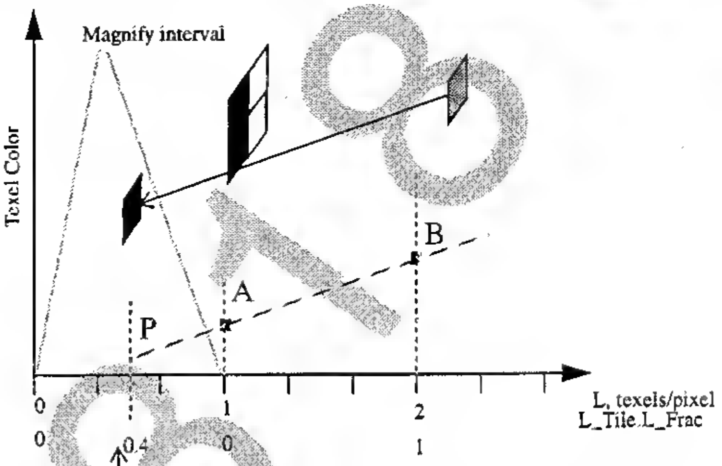
If the (power of 2) texture matches along the 4 edges, clamp can be turned off and the bilerp filter will use the texel from the other edge of the wrapping texture to filter to.

Note: Since point sampled and bilerp modes cause a shift of 0.5 texels in the displayed primitive, to switch between point sampled and bilerp modes without shifting the texture one of the following methods may be used: 1) use a different primitive with a 0.5 shift in the texture coordinates; 2) Set the 0.5 texel shift in SL and TL in the texture tile (SL and TL are subtracted from the incoming texture coordinates)

Note: If the mxn texture is too large to fit in tmem, the polygon and the texture can be broken up along u,v and s,t in appropriately sized tiles. For the bilerp to work along the tile boundaries, an extra row (or column) of texels around each tile border needs to be loaded i.e the resulting polygons will be disjoint but each tile (that is not a border tiles) will have an overlap of 2 texels with any adjacent tile.

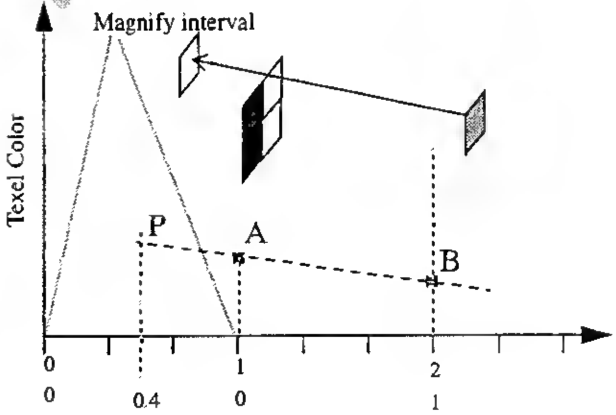
960 2458

Figure 14-8 Sharpen Extrapolation


Magnify interval
Texel Color
B
A
P
0
1
2
L, texels/pixel
L_Tile.L_Frac
0
0.4
0
1

The change in color between texel A and B is extrapolated using the equation P = A + (B-A)*(Lfrac-1.0) Notice that the extrapolation makes the dark texel even darker...

and light texels become lighter after the extrapolation, thus enhancing the apparent sharpness of the edge.


Magnify interval
Texel Color
P
A
B
0
1
2
0
0.4
0
1

11573189

1157389

Texture Memory

Memory Organization

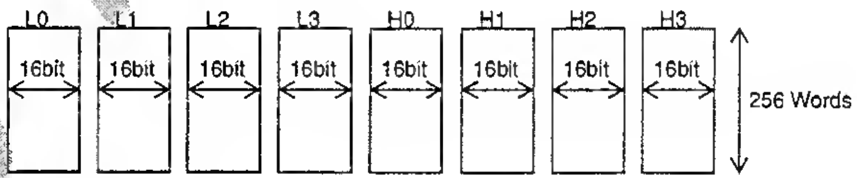
Because texturing requires a large amount of random accesses with consistent access time to texture memory, it is impractical to texture directly from DRAM. The approach taken by the Nintendo64 system is to cache up to 4 KB of an image in an on-chip, high-speed texture memory called Tmem. All primitives are textured using the contents of Tmem. The basic sequence of events needed to texture a primitive is:

1. Load a texture tile into Tmem.
2. Describe attributes of the texture tile.
3. Render primitives that use this tile.

Tmem should indeed be considered a cache from the programmer's point of view. Since each tile must be loaded from DRAM, it makes sense to render as many primitives as possible, using the current tile before loading the next one in order to conserve DRAM bandwidth.

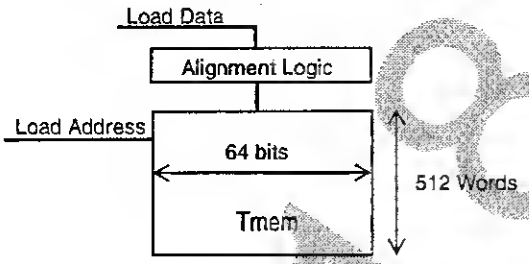
Physically, Tmem is arranged as shown in Figure 14-9. L0-3 are referred to as the low half of Tmem, H0-3 are referred to as the high half of Tmem.

Figure 14-9 Physical Tmem Diagram



For loading, Tmem is arranged logically, as shown in Figure 14-10.

Figure 14-10 Tmem Loading


Load Data
Alignment Logic
Load Address
64 bits
Tmem
512 Words

The following table shows the maximum tile sizes that can be stored in the 4KB Texture Memory. Images larger than this will be tiled.

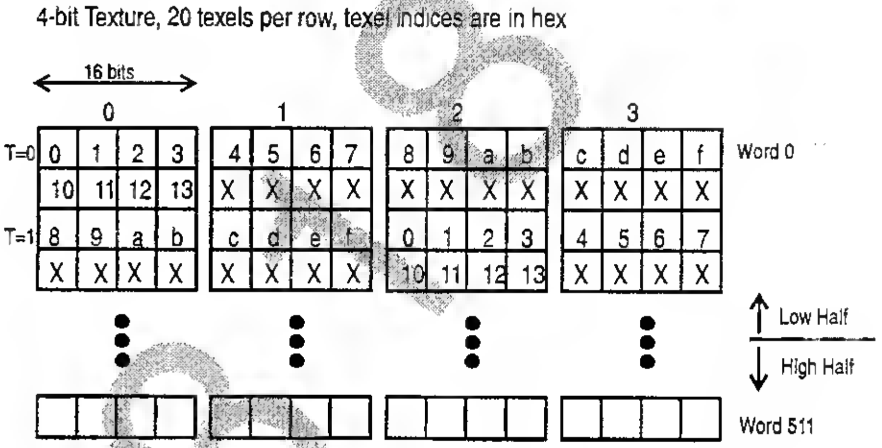
Table 14-8 Maximum tile sizes in TMEM

Texel Type	Maximum Texel Count
4-bit (I, IA)	8K
4-bit Color Index	4K (plus 16 palettes)
8-bit (I, IA)	4K
8-bit Color Index	2K (plus 256-entry LUT)
16-bit RGBA	2K
16-bit IA	2K
16-bit YUV	2K Y's, 1K UV pairs
32-bit RGBA	1K

1157389

Four-bit textures are stored in Tmem as shown, as shown in Figure 14-11.

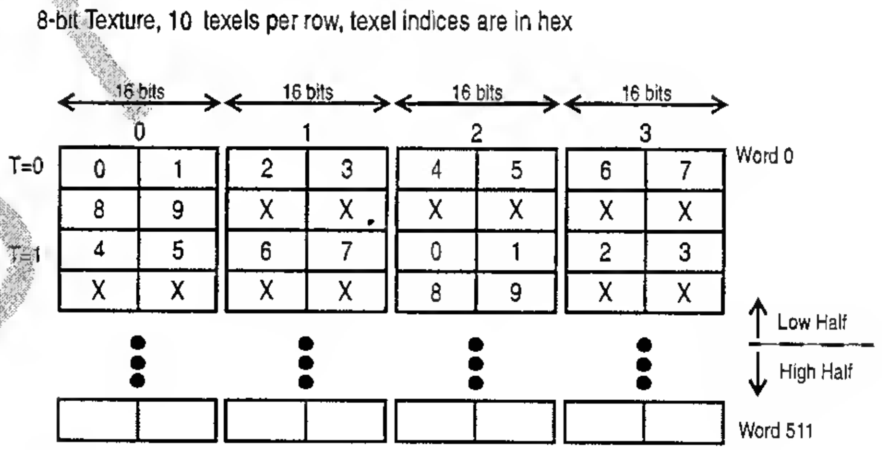
Figure 14-11 Four-Bit Texel Layout in Tmem


4-bit Texture, 20 texels per row, texel indices are in hex
16 bits
0
1
2
3
T=0
T=1
Word 0
Low Half
High Half
Word 511

	0				1				2				3				
T=0	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Word 0
	10	11	12	13	X	X	X	X	X	X	X	X	X	X	X	X	
T=1	8	9	a	b	c	d	e	f	0	1	2	3	4	5	6	7	
	X	X	X	X	X	X	X	X	10	11	12	13	X	X	X	X	

Eight-bit textures are stored in Tmem, as shown in Figure 14-12.

Figure 14-12 Eight-Bit Texel Layout in Tmem

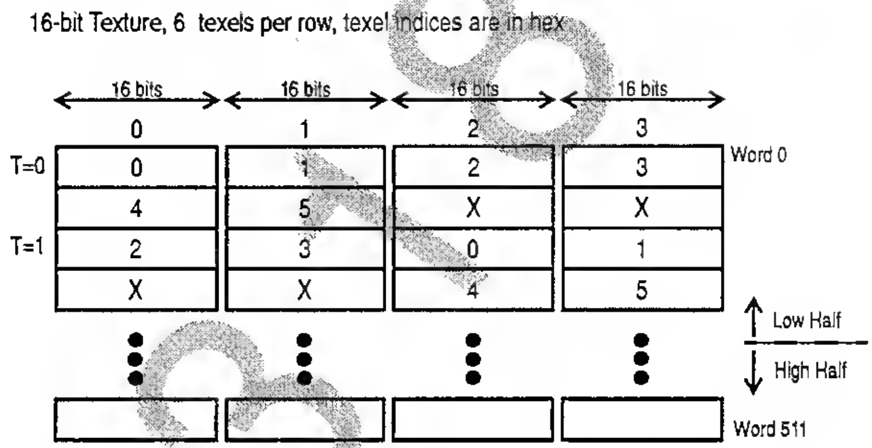

8-bit Texture, 10 texels per row, texel indices are in hex
16 bits
16 bits
16 bits
16 bits
0
1
2
3
T=0
T=1
Word 0
Low Half
High Half
Word 511

	0		1		2		3		
T=0	0	1	2	3	4	5	6	7	Word 0
	8	9	X	X	X	X	X	X	
T=1	4	5	6	7	0	1	2	3	
	X	X	X	X	8	9	X	X	

1157389

Sixteen-bit textures (except YUV) are stored in Tmem, as shown in Figure 14-13.

Figure 14-13 Sixteen-Bit Texel Layout in Tmem

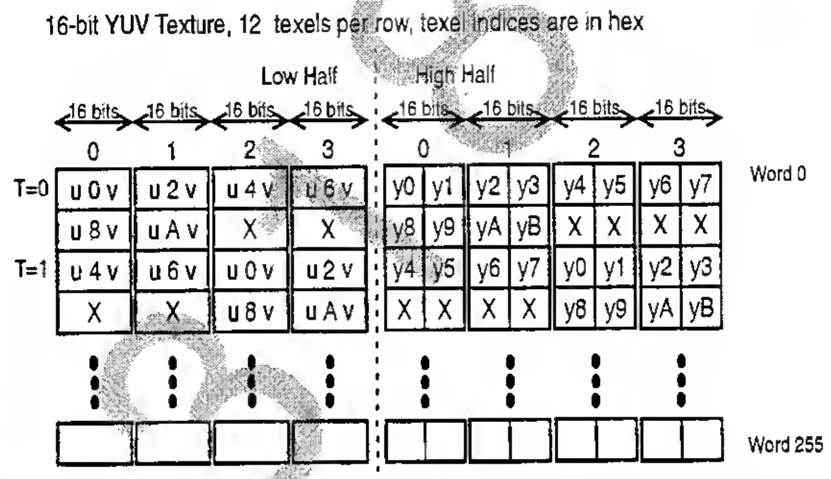

16-bit Texture, 6 texels per row, texel indices are in hex
16 bits
16 bits
16 bits
16 bits
0
1
2
3
T=0
0
1
2
3
Word 0
4
5
X
X
T=1
2
3
0
1
X
X
4
5
Low Half
High Half
Word 511

Sixteen-bit YUV textures are stored in Tmem, as shown in Figure 14-14. Note that YUV texels must be loaded in pairs. In other words two Y's at a time. Also note that if filtering is enabled, an additional UYVY pair must be

11573189

loaded per row and SH set accordingly to allow proper filtering of the last UV texel per row.

Figure 14-14 YUV Texel Layout in Tmem

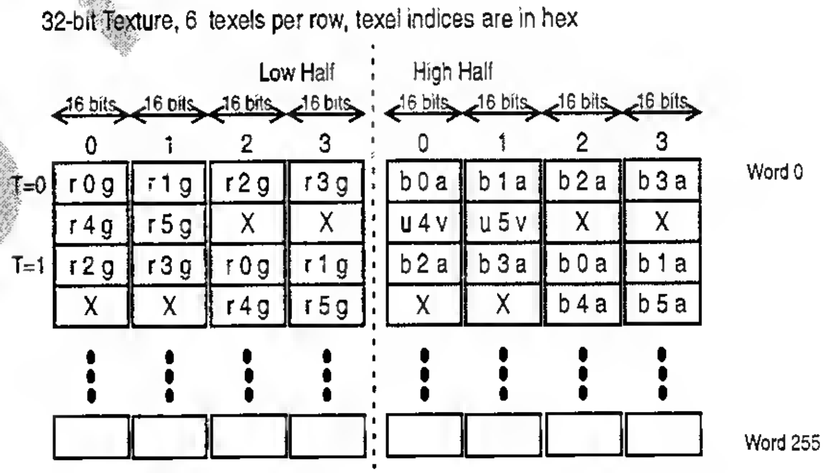


16-bit YUV Texture, 12 texels per row, texel indices are in hex

	0	1	2	3	0		1		2		3		
T=0	u 0 v	u 2 v	u 4 v	u 6 v	y0	y1	y2	y3	y4	y5	y6	y7	Word 0
	u 8 v	u A v	X	X	y8	y9	yA	yB	X	X	X	X	
T=1	u 4 v	u 6 v	u 0 v	u 2 v	y4	y5	y6	y7	y0	y1	y2	y3	
	X	X	u 8 v	u A v	X	X	X	X	y8	y9	yA	yB	
	⋮	⋮	⋮	⋮	⋮		⋮		⋮		⋮		
													Word 255

Thirty-two bit (RGBA) textures are stored in Tmem, as shown in Figure 14-15.

Figure 14-15 Thirty-Two Bit RGBA Texel Layout in Tmem



32-bit Texture, 6 texels per row, texel indices are in hex

	0	1	2	3	0	1	2	3	
T=0	r 0 g	r 1 g	r 2 g	r 3 g	b 0 a	b 1 a	b 2 a	b 3 a	Word 0
	r 4 g	r 5 g	X	X	u 4 v	u 5 v	X	X	
T=1	r 2 g	r 3 g	r 0 g	r 1 g	b 2 a	b 3 a	b 0 a	b 1 a	
	X	X	r 4 g	r 5 g	X	X	b 4 a	b 5 a	
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
									Word 255

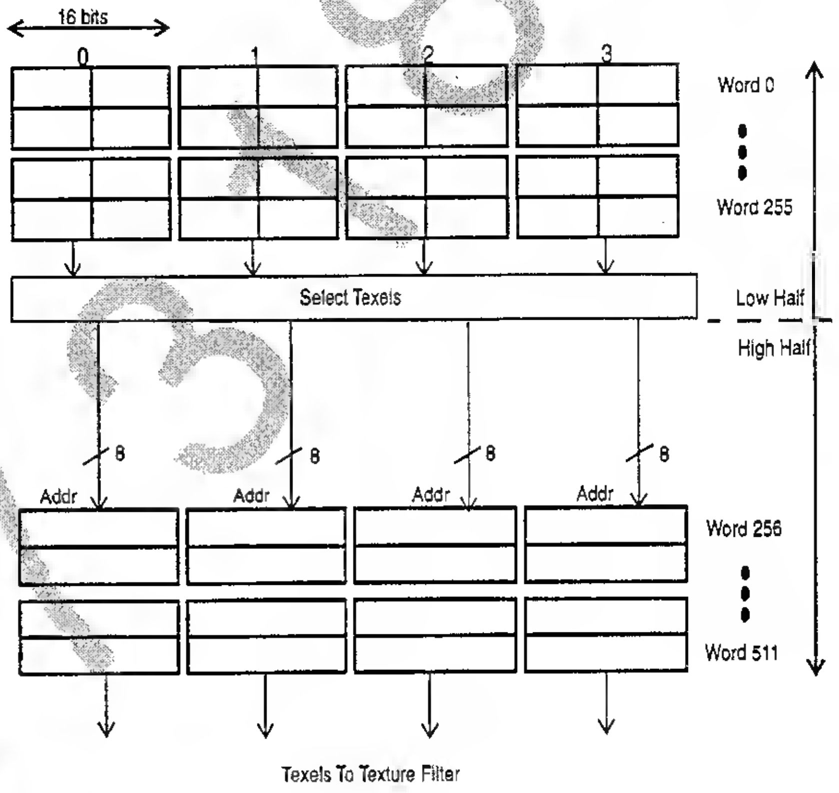
1157389

For color index (CI) textures, the texture is stored in the lower half of Tmem. and the Texture/Color Look-Up Table (TLUT) is stored in the upper half of Tmem. For 4-bit CI textures, the texels (or indices) addressed in the lower half of Tmem have the 4-bit palette number for the tile prepended to create an 8-bit address into the upper half of Tmem. Since four texels are addressed simultaneously, there must be four (usually identical) TLUTs stored in the upper half of Tmem across the four banks.

For 4-bit CI textures, the palette effectively selects one of sixteen possible tables, each table having sixteen entries. Each table is aligned on 16-word boundaries. Note that there are two choices for the texel type that resides in the TLUT: 16-bit RGBA, or 16-bit IA. The type is selected using the *gDPSetTextureLUT()* command. This command also configures the Tmem as shown in Figure 14-16. Because of this, CI textures cannot be combined with other texture types in two-cycle mode.

115739

Figure 14-16 Tmem Organization for Eight-Bit Color Index Textures


16 bits
0
1
2
3
Word 0
Word 255
Select Texels
Low Half
High Half
8
8
8
8
Addr
Addr
Addr
Addr
Word 256
Word 511
Texels To Texture Filter

Eight-bit CI textures do not use the palette number of the tile, since they address the whole 256 TLUT directly. It is possible to use the 8-bit mode for storing index textures that have between 16 and 256 entries.

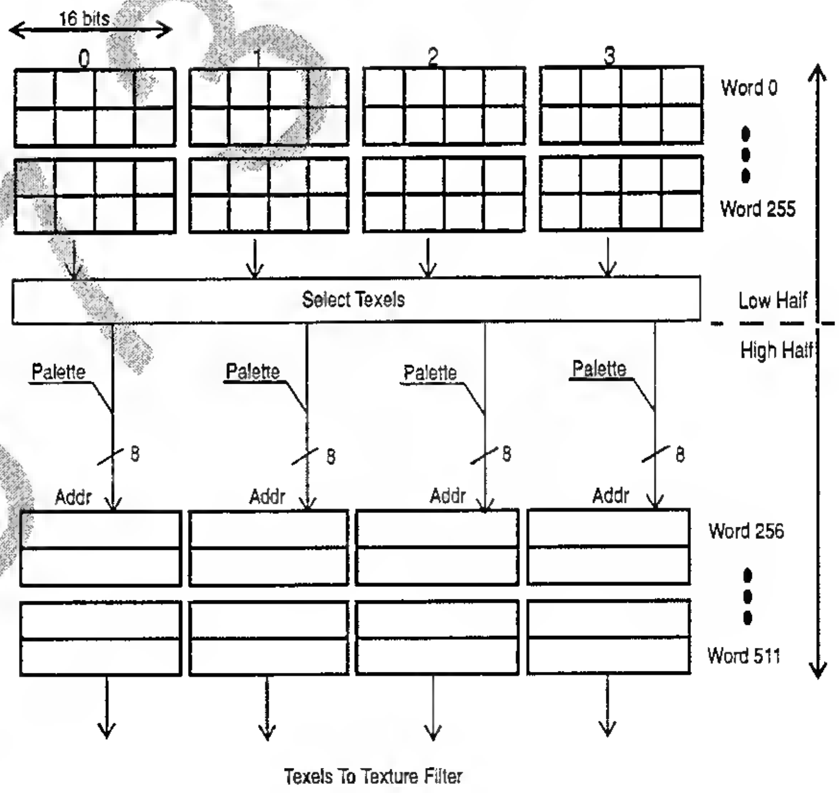
For example, you could define a texture that had 40 entries, numbered 0-39, and load the TLUT into the upper half of Tmem (word 256). Further suppose that you had another texture with indices 40-69. You could load this texture's 30 entry TLUT into Tmem, starting at word 296.

1157389

Assuming that both textures together fit into the lower half of Tmem (2 KB), these textures could be co-resident in Tmem. It is also possible to have CI textures co-resident with other non-CI textures.

In the above example, you are using only the first 70 words of upper Tmem for TLUTs. You could use the remaining 186 words to store a 4-bit I texture, for example. Note that even though you can store CI and other types together in Tmem, you cannot access these types simultaneously in two-cycle mode, because the configuration of the Tmem for CI textures is controlled with a mode bit that must be updated using the *gDPSetTextureLUT* command, as mentioned previously.

Figure 14-17 Tmem Organization for Four-Bit CI textures


16 bits
0
1
2
3
Word 0
Word 255
Select Texels
Low Half
High Half
Palette
Palette
Palette
Palette
8
8
8
8
Addr
Addr
Addr
Addr
Word 256
Word 511
Texels To Texture Filter

1151319

11573169

Texel Formatting

In the RDP graphics pipeline, most operations are done on 8-bit-per-component RGBA pixels. After looking up the texels, the texture unit converts them into the 32-bit RGBA format. Table 14-9 describes how each type is converted. The format for beatified descriptions is [MSB:LSB] where MSB is the most significant bit and LSB is the least significant bit. Bit fields are grouped together in braces {} with the most significant field on the left and the least significant field on the right.

Table 14-9 Texel Output Formatting

Type	Size	Input Format	Output Format			
			Red	Green	Blue	Alpha
I	4	I[3:0]	{[3:0], [3:0]}	{[3:0], [3:0]}	{[3:0], [3:0]}	{[3:0], [3:0]}
I	8	I[7:0]	[7:0]	[7:0]	[7:0]	[7:0]
IA	4	I[3:1], A[0]	{[3:1], [3:1], [3:2]}	{[3:1], [3:1], [3:2]}	{[3:1], [3:1], [3:2]}	255*[0]
IA	8	I[7:4], A[3:0]	{[7:4], [7:4]}	{[7:4], [7:4]}	{[7:4], [7:4]}	{[3:0], [3:0]}
IA	16	I[15:8], A[7:0]	[15:8]	[15:8]	[15:8]	[7:0]
RGBA	16	R[15:11], G[10:6], B[5:1], A[0]	{[15:11], [15:13]}	{[10:6], [10:8]}	{[5:1], [5:3]}	255*[0]
RGBA	32	R[31:24], G[23:16] , B[15:8], A[7:0]}	[31:24]	[23:16]	[15:8]	[7:0]

Texture Loading

Loading a texture actually consists of several steps. Internally, the RDP treats loading a texture as if it were rendering a textured rectangle into Tmem. To load a texture, you must describe the texture tile to be loaded, render (load) it into Tmem, and describe the tile to be rendered. An important consequence of these steps is that you can load a texture in one way and render it in completely different way.

For example, the GBI macro *gsDPLoadTextureTile* performs all the tile and load commands necessary to load a texture tile. The sequence of commands is shown below (macros shown without parameters):

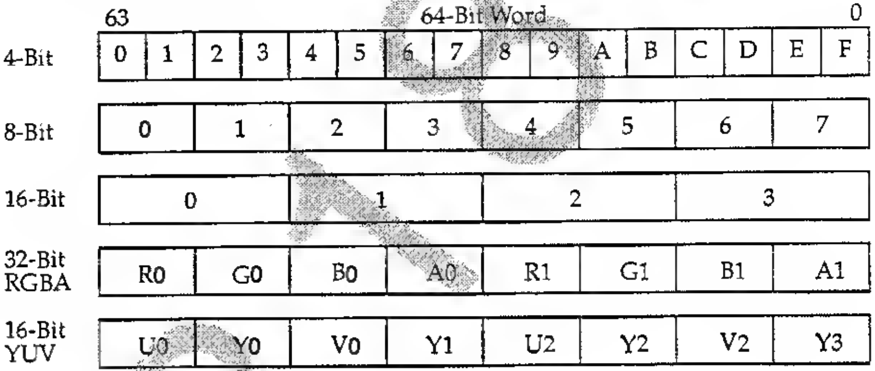
```
gsDPSetTextureImage
gsDPSetTile /* G_TX_LOADTILE */
gsDPLoadSync
gsDPLoadTile /* G_TX_LOADTILE */
gsDPSetTile /* G_TX_RENDERTILE */
gsDPSetTileSize /* G_TX_RENDERTILE */
```

This sequence of commands loads a texture tile using the tile descriptor G_TX_LOADTILE (tile 7) and renders using G_TX_RENDERTILE (tile 0). Since the tile descriptor used to load the tile is different from the one used to render the texture, there is no possibility of tile usage conflict, so a *TileSync* command is unnecessary. The *TileSync* command is used in situations where you may want to use the same tile for both loading and rendering a texture. It basically inserts a bubble in the RDP pipeline to guarantee that the load tile descriptor isn't changed by the render tile before the load is actually done.

The *gsDPSetTextureImage* command sets a pointer to the location of the image. Then the *gsDPSetTile* is used to indicate where in Tmem you want to place the image, how wide each line is, and the format and size of the texture. A *gsDPLoadSync* command makes sure that any previous load is completely finished before this texture is loaded. Then the actual *gsDPLoadTile* command is issued, which loads the texture into Tmem. The final *gsDPSetTile* and *gsDPSetTileSize* are used to set the tile descriptors correctly for the tile used when rendering.

The textures are stored big-endian in memory and should obey the following format for a 64-bit word in memory.

Figure 14-18 Texel Formats in DRAM

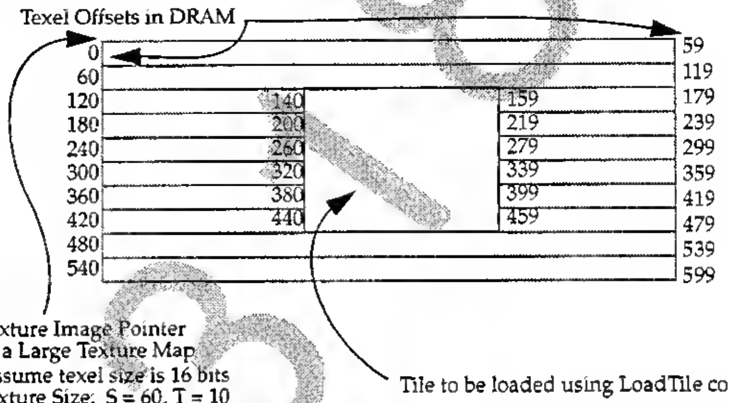

63
64-Bit Word
0
4-Bit
0 1 2 3 4 5 6 7 8 9 A B C D E F
8-Bit
0 1 2 3 4 5 6 7
16-Bit
0 1 2 3
32-Bit RGBA
R0 G0 B0 A0 R1 G1 B1 A1
16-Bit YUV
U0 Y0 V0 Y1 U2 Y2 V2 Y3

1157309

Load Tile

The *LoadTile* command allows a programmer to load an arbitrary rectangular region of a larger texture in DRAM into Tmem. The following examples assume a 16-bit texel type.

Figure 14-19 Example of LoadTile Command Parameters



Texture Image Pointer
to a Large Texture Map
Assume texel size is 16 bits
Texture Size: S = 60, T = 10

Tile to be loaded using LoadTile command

Line: $\frac{\text{20 texels/line * 2 bytes/texel}}{\text{8 bytes/word}}$

SL = 20, TL = 2, SH = 39, TH = 7

Each line of the tile, for example, texels 140-159, will be at least one DRAM transfer. The advantage of LoadTile is that you can load arbitrary tiles from a larger map.

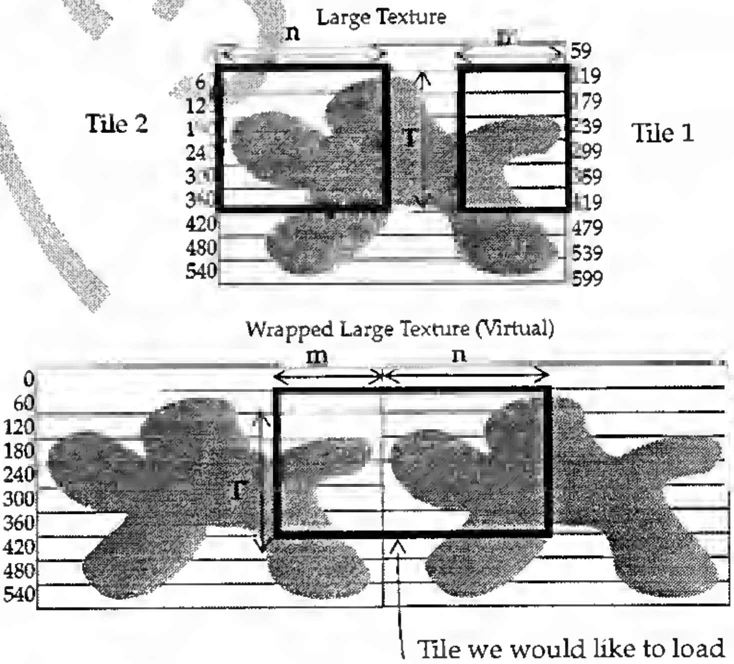
When textures are loaded as a tile, it means that (at least) each line of the texture is a separate DRAM transfer. Each line's transfer may be broken into multiple smaller transfers, depending on how big it is and whether it crosses DRAM page boundaries. Since the DRAMs are block transfer type devices, there is a fixed amount of overhead for each transfer, so long transfers are desirable. For this reason, you should try to load your texture using the longest dimension of the tile. Also, each line of a tile is padded automatically to Tmem word (64-bit) boundaries. If your tile line size is not a multiple of 64-bits, some Tmem space is being wasted. Also when tiling a larger texture image into multiple tiles, an extra row and column are usually loaded to allow proper filtering of the texels along the border of the tile (to avoid seams).

Note: The RDP commands LoadTile, LoadBlock, and LoadTLUT set the tile parameters SL,TL,SH,TH when they are executed. After the load command, it may be necessary to use the SetTileSize command to restore these parameters if you want parameters other than were used in the Load command. In the gbi.h texture load macros, the SetTileSize command is always used following a Load command.

Wrapping a Large Texture Using Load Tile

It is possible to effectively 'wrap' large textures (textures too large to fit entirely in Tmem) by careful loading using the *LoadTile* command. There are (at least two) methods for doing this. Figure 14-20, "Wrapping a Large Texture Using Two Tiles," on page 251 shows a large texture in memory. We want to load a tile as if the texture had been wrapped in the S direction, and the tile straddles the wrap region.

Figure 14-20 Wrapping a Large Texture Using Two Tiles


Large Texture
n
m
Tile 2
Tile 1
T
6
12
24
420
480
540
59
179
479
539
599
Wrapped Large Texture (Virtual)
m
n
0
60
120
180
240
300
360
420
480
540
T

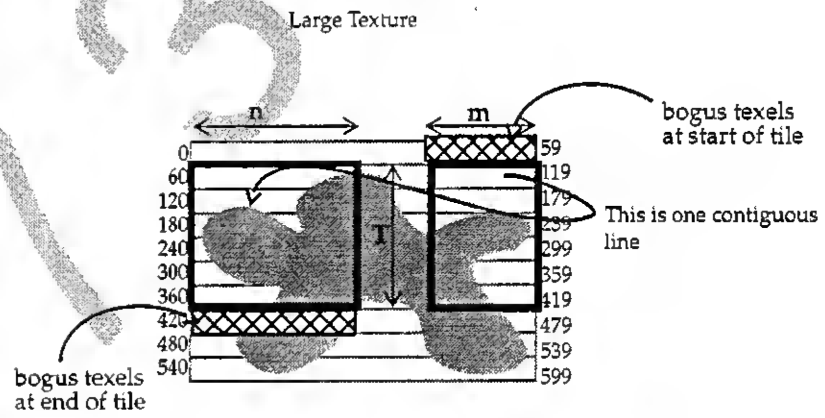
Tile we would like to load

1157389

One way to effectively load the wrapped tile is to actually load two interleaved tiles. To interleave two tiles in Tmem, load tile 1 but set the tile's *Line* parameter to n+m Tmem words, where n is the number of words in a line of Tile 1 and m is the number of words in tile 2. SL,SH,TL,TH should be set to load Tile 1. Now load Tile 2, setting the tile's *Tmem Address* to n. Also set the SL,TL,SH,TH for Tile 2. After the loads, reset the render tile's Tmem Address to 0. Set SL,SH,TL,SH to be the total composite tile size. Note that only Tile 1's width must be a multiple of Tmem words. Tile 2's width can be any number of texels and the remainder of the last Tmem word for each line will simply be undefined.

Another, possibly more straightforward method, relies on the fact that at the end of each line of the large texture, the addresses will naturally roll into the next line.

Figure 14-21 Wrapping a Large Texture Using One Tile

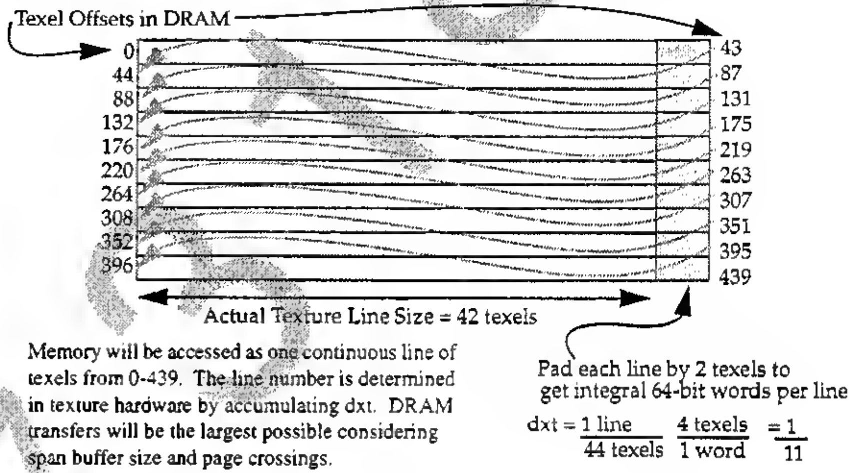

Large Texture
n
m
bogus texels at start of tile
0
60
120
180
240
300
360
420
480
540
59
119
179
239
299
359
419
479
539
599
T
This is one contiguous line
bogus texels at end of tile

So, as shown in Figure 14-21, "Wrapping a Large Texture Using One Tile," on page 252, you can load a single tile starting at address 60 minus **m** words. The tile's *Line* parameter should equal **m+n**. Set the *Tmem Address* parameter to 0 during the load. Make sure to load **T+1** lines. After the load, set *Tmem Address* to **m**, and set the *SL,SH,TL,TH* to the actual tile size. This method wastes **m** words at the beginning of Tmem and n words at the end of Tmem but has the advantage of using only one load.

Load Block

A more memory-bandwidth efficient way to load textures is the *LoadBlock* command. This command essentially treats each texture as a single long line of data. This allows the MI to transfer the maximum amount of data for each transfer.

Figure 14-22 Example of LoadBlock Command Parameters


Texel Offsets in DRAM
0
44
88
132
176
220
264
308
352
396
43
87
131
175
219
263
307
351
395
439
Actual Texture Line Size = 42 texels
Memory will be accessed as one continuous line of texels from 0-439. The line number is determined in texture hardware by accumulating dxt. DRAM transfers will be the largest possible considering span buffer size and page crossings.
Pad each line by 2 texels to get integral 64-bit words per line
dxt = 1 line / 44 texels · 4 texels / 1 word = 1 / 11

The *LoadBlock* command uses the parameter *dxt* to indicate when it should start the next line. *Dxt* is basically the reciprocal of the number of words (64-bits) in a line. The texture coordinate unit increments a counter by *dxt* for each word transferred to Tmem. When this counter rolls over into the next integer value, the line count is incremented. The line count is important because the data in odd lines is swapped to allow interleaved access when rendering. This works great when dxt is a power of two. However, if dxt is not a power of two, the line counter can be corrupted due to accumulated error. Appendix A contains a table that indicates how many lines for a certain size can be in a load block for a tile before the line count is corrupted.

It is possible to load a set of texture tiles using a single *LoadBlock* command (MIP maps, for example). However, if the tiles have different widths, the single *dxt* parameter is not enough to do proper interleaving. In these cases, the data must be pre-interleaved and the *dxt* parameter should be set to zero.

The *LoadTlut* command is an efficient way of loading texture look-up tables into the high half of TMEM. System memory is conserved using this command as each 16-bit color value is "quadricated" as it is read in and written to the TMEM. In other words, it isn't necessary to store four times the data in memory. The load hardware will expand it out into a 64-bit word during the load. This saves system memory as well as memory bandwidth. Two types of TLUTs are supported: 16-bit RGBA and 16-bit IA. TLUT depth can range from 16 words (4-bit CI) to 256 words (8-bit CI). *LoadTile* or *LoadBlock* can still be used for loading the TLUT however the data will have to be quadricated in system memory first.

Loading Notes

4-bit types should be loaded as 16-bit types and then rendered as 4-bit types. This does not restrict 4-bit types in any way and still allows for rows with an odd number of 4-bit texels.

When using *LoadBlock*, no more than 2048 texels can be loaded at once. So for example if you wanted to load 4K 8-bit texels, load them as 2K 16-bit texels and then render them as 8-bit texels. If you're using 16-bit or 32-bit there is no need for a special case since TMEM cannot hold more than 2K 16-bit or 1K 32-bit texels.

To improve performance by minimizing the number of syncs required, the user can interleave the tile loads and renders with different tile indices. For example, load using tile 7 while rendering using tile 0.

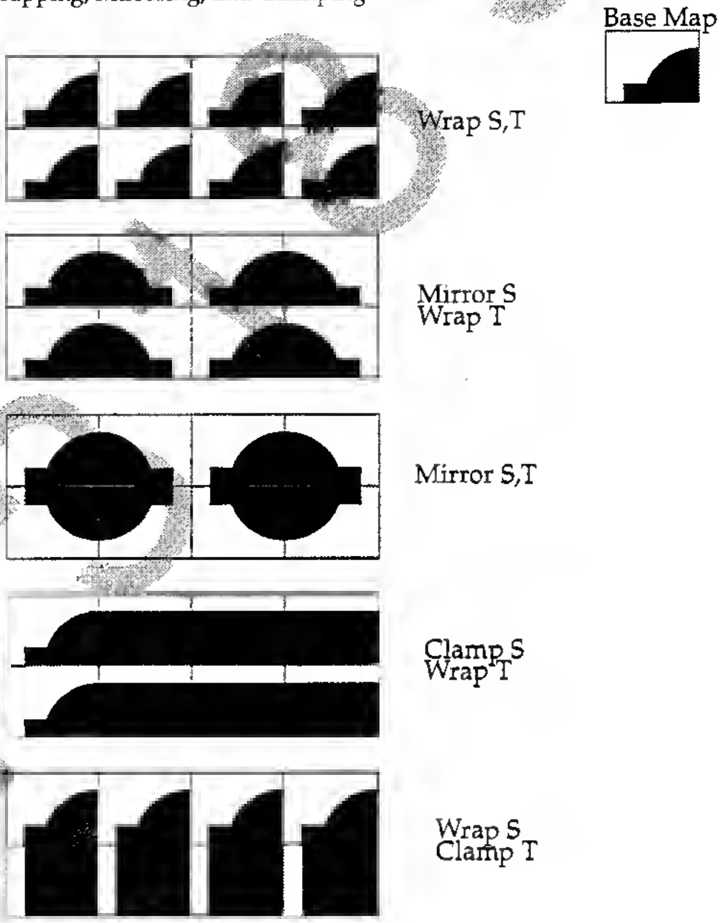
1151389

Examples

After texture coordinates are converted to Tile Space, they may be wrapped, clamped, or mirrored. Figure 14-23 shows how wrapping, mirroring, and clamping affect the tile-relative coordinates. The S and T coordinates have independent controls for wrapping, mirroring, and clamping.

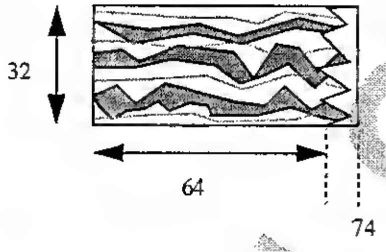
115731899

Figure 14-23 Wrapping, Mirroring, and Clamping


Base Map
Wrap S,T
Mirror S
Wrap T
Mirror S,T
Clamp S
Wrap T
Wrap S
Clamp T

1157909

Figure 14-24 Wrapping Within a Texture Tile



Textured log using 3 textured cylinders. The middle cylinder sets the tile's mask to 6 so that the texture wraps every 64 texels. The end cylinders set the tile's clamp bit and have coordinates that access the jagged part of the texture. Advantages include easier modeling, use of one load command, and possibly tighter Tmem packing than if two separate textures were used.

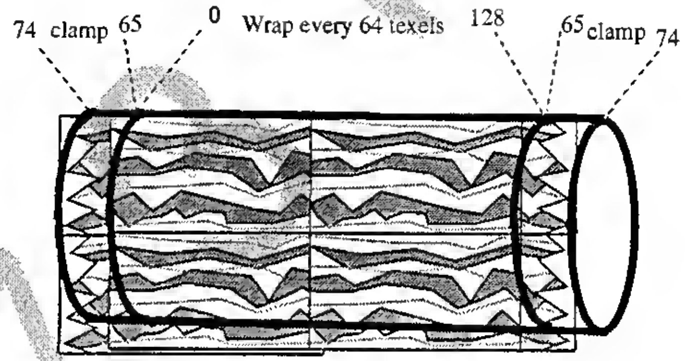
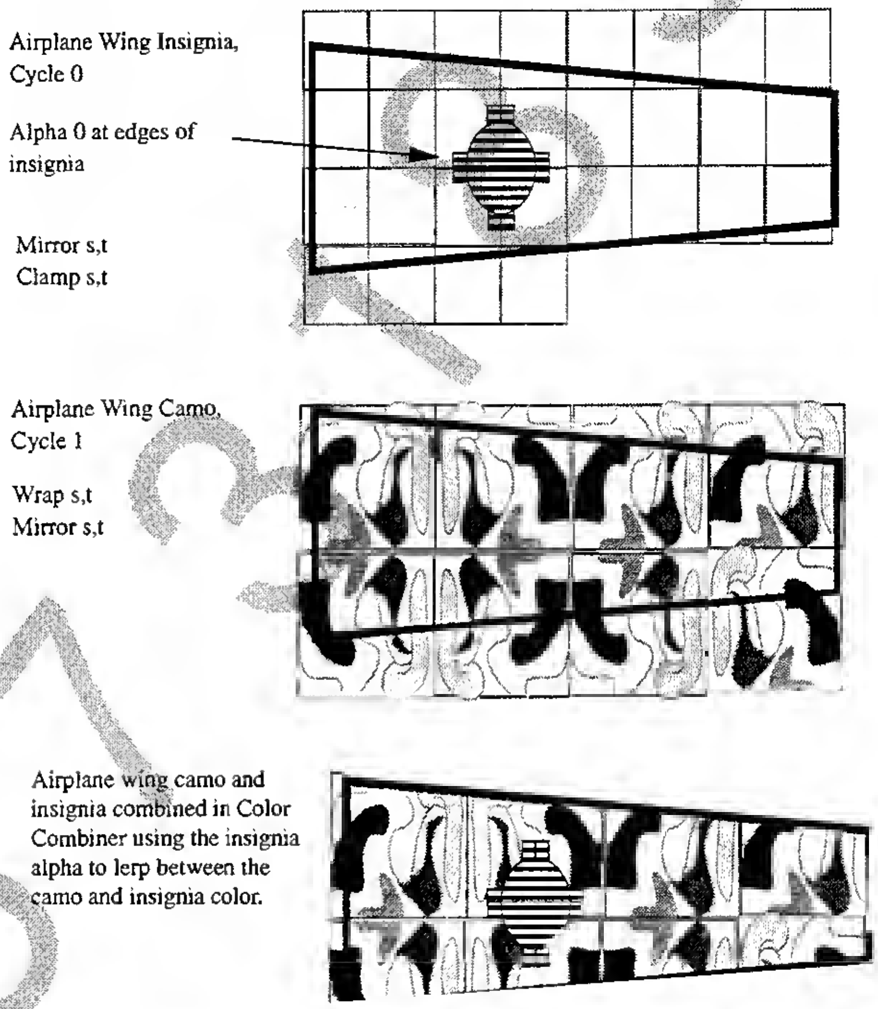

74 clamp 65
0
Wrap every 64 texels
128
65 clamp 74

Figure 14-25 Example of Texture Decals


Airplane Wing Insignia,
Cycle 0
Alpha 0 at edges of
insignia
Mirror s,t
Clamp s,t
Airplane Wing Camo,
Cycle 1
Wrap s,t
Mirror s,t
Airplane wing camo and
insignia combined in Color
Combiner using the insignia
alpha to lerp between the
camo and insignia color.

11573189

Restrictions

Texture Types and Modes

The following is a list of restrictions concerning the use of certain textures types in certain modes:

Point Sample

Clamp &/ | (wrap | mirror) works for all texel types.

Filter

Clamp works for all texel types. Wrap t | mirror t | (clamp t & wrap t) | (clamp t & mirror t) works for all texel types.

Wrap s | mirror s | (clamp s & wrap s) | (clamp s & mirror s) works for all texel types **except YUV.**

Copy

Clamping is implicitly disabled for copy mode. 32-bit RGBA and YUV texel types **are not supported.** To copy these types, they should be loaded and copied as 16-bit RGBA type texels. When using a 16-bit RGBA type to copy a 32-bit RGBA or YUV texture, mirroring **in s is not supported.**

Wrap or mirror works for 4, 8, and 16-bit types.

LOD

You must put the RDP in two-cycle mode to use texture LOD.

Alignment

The texture image pointer, as defined using the *gDPSetTextureImage* command, must be 8-byte aligned. Additionally, each tile must be aligned according to its size. For example, 8-bit texture tiles must be aligned to 8-bit boundaries, 16-bit textures to 16-bit boundaries, etc. **One exception is 4-bit tiles, which must be accessed on byte (8-bit boundaries).**

Tiles

The maximum size of a tile is 256 rows (t coordinate) and 1024 texels (s coordinate) within the limits of Tmem size. It is better to always make the s coordinate the longer coordinate in terms of load performance.

You should avoid shifting coordinates left using the shift parameter of a tile unless necessary. See the example under Multiple Tile Effects in the Applications section.

Coordinate Range

The valid texture coordinate range is currently from -1024.0 to +1023.99. A total range of 2K texels across a primitive. The texture hardware can handle this full range without any noticeable loss of accuracy. For small coordinate ranges however, if given a choice of coordinates close to zero or coordinates close to 1024, slightly higher quality may result from the lower coordinates.

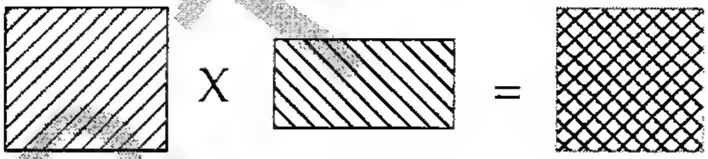
11578139

Applications

Multiple Tile Effects

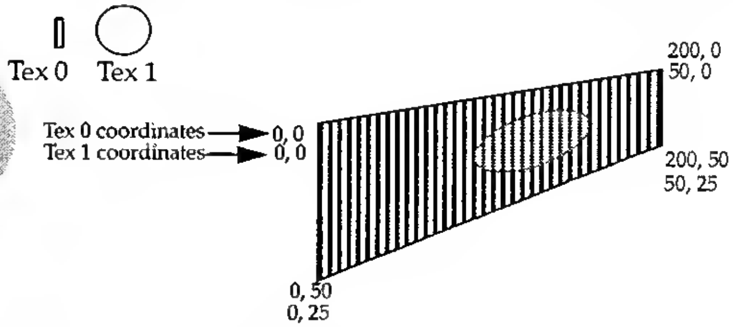
Interference Textures

Since you can access two separate tiles in two-cycle mode, it easy to achieve interference pattern effects. Of course, you can use textures that are different sizes (wrap on different intervals) to decrease the amount of apparent repetition. This is especially useful for textures on terrain or for waves on the ocean, for example.



Lighting with Textures

Multiple tiles can be used for lighting effects. In the example below, a small texture is repeated many times but a small light texture is scaled up to create the effect of a spotlight. In this example you could use the input coordinates should be defined using Tex 0's coordinates. The *shift* parameter of the tile descriptor for Tex 1 could be used to right shift the input coordinates to the required values. It would be a bad idea to use Tex 1's coordinates as the



input coordinates and then left shift to obtain Tex 0's coordinates. This is because when you shift left, you shift zeros into the lsb's of the coordinate, thus losing precision.

Extended Alpha Using Multiple Textures

The 16 bit RGBA texture type is often used to texture sprites and billboards because this is the only type that allows a large number of colors. Unfortunately, this type only has one bit of alpha (which means you cannot prefilter texture edges), and can lead to pixelated texture edges.

One way to get more bits of alpha (in order to create smoother outlines) is to use two tiles. The first tile describes the RGB color of the texture, while the second tile describes the alpha channel of the texture. Render the texture in two-cycle mode. In the color combiner, select T0 as the source and in the alpha combiner select T1 as the source.

A code fragment indicates how to set the combine modes and load the textures:

```
#define MULTIBIT_ALPHA  0, 0, 0, TEXEL0, 0, 0, 0, TEXEL1

...
    /* use special combine mode */
    gsDPSetCombineMode(MULTIBIT_ALPHA, G_CC_PASS2),
...
    /*
     *  Load alpha texture at Tmem = 256, notice I use a
     *  different load macro that allows specifying Tmem
     *  address.
     */
    _gsDPLoadTextureBlock_4b(I4molecule, 256, G_IM_FMT_I,
            32, 32, 0,
             G_TX_WRAP, G_TX_WRAP,
             5, 5, G_TX_NOLOD, G_TX_NOLOD),

    /*
     *  Load color texture starting at Tmem=0
     */
    gsDPLoadTextureBlock(RGBA16molecule, G_IM_FMT_RGBA,
            G_IM_SIZ_16b, 32, 32, 0, G_TX_WRAP, G_TX_WRAP,
             5, 5, G_TX_NOLOD, G_TX_NOLOD),
```

1157389

```
    /*
     *  Since normal load macros use tile 0 for render, I
     *  need to set tile 1 manually to point at alpha
     *  texture.
     */

    gsDPSetTile(G_IM_FMT_I, G_IM_SIZ_4b, 2, 256, 1,
        0,
        0, 0, 0,
        0, 0, 0),
    gsDPSetTileSize( 1, 0, 0, 31 << 2, 31 << 2),

...
    /* make sure in two-cycle mode */
    gsDPSetCycleType(G_CYC_2CYCLE),
```

11573189

Appendix A: LoadBlock Line Limits

The table below lists the maximum number of lines that can be properly transferred for a given texture width.

Note: The *absolute max lines* column refers to the number of lines that *could* be transferred if only limited by Tmem size. If *absolute max lines* field is empty, it indicates that the *max lines* was equal to *absolute max lines*. If *max lines* is empty it indicates that zero lines could be transferred correctly using these parameters.

This table only applies to 16-bit texels.

Table 14-10 Limits on Number of Lines for LoadBlock Command

Width (16b texels)	Max Lines	Absolute Max Lines
4	512	
8	256	
12	170	
16	128	
20	102	
24	85	
28	73	
32	64	
36	56	
40	51	
44	20	46
48	42	
52	26	39
56	14	36
60	19	34

Table 14-10 Limits on Number of Lines for LoadBlock Command

Width (16b texels)	Max Lines	Absolute Max Lines
64	32	
68	13	30
72	28	
76	26	
80	8	25
84	9	24
88	4	23
92	4	22
96	5	21
100	20	
104	13	19
108	18	
112	3	18
116	6	17
120	3	17
124	2	16
128	16	
132-136	2	15
140	3	14
144	14	
148	2	13
152	13	
156	2	13

11573189

Table 14-10 Limits on Number of Lines for LoadBlock Command

Width (16b texels)	Max Lines	Absolute Max Lines
160	1	12
164	12	
168	4	12
172-184	2	11
188-192	2	10
196	4	10
200	10	
204	---	10
208	1	9
212	2	9
216	9	
220	---	9
224	1	9
228	8	
232	---	8
236	2	8
240	---	8
244	1	8
248	---	8
252	1	8
256	8	
260-264	---	7
268	1	7

11573189

Table 14-10 Limits on Number of Lines for LoadBlock Command

Width **(16b texels)**	**Max Lines**	**Absolute** **Max Lines**
272	---	7
276	1	7
280	---	7
284	2	7
288-292	---	7
296	1	6
300	---	6
304	6	
308-312	---	6
316	4	6
320-324	---	6
328	6	
332-340	---	6
344	1	5
348-356	---	5
360	1	5
364-372	---	5
376	1	5
380-388	—	5
392	2	5
396-408	---	5
412	1	4
416-428	---	4

11573189

Table 14-10 Limits on Number of Lines for LoadBlock Command

Width (16b texels)	Max Lines	Absolute Max Lines
432	4	
436-452	—	4
456	4	
460-480	—	4
484	1	4
488-508	—	4
512	4	
516-544	—	3
548	2	3
552-584	—	3
588	1	3
592-628	—	3
632	2	3
636-680	—	3
684	2	
688-744	—	2
748	1	2
752-816	—	2
820	2	
824-908	—	2
912	2	
916-1020	—	2

11573189

11573189

Chapter 15

Texture Rectangles (Hardware Sprites)

Warning: **Code fragments in this chapter have not been fully verified. A demo containing these examples will be included in a future software release.**

A texture rectangle is a special primitive supported by the Reality Display Processor (RDP) hardware. This primitive is intended to provide simple 'sprite' capabilities with a minimum number of parameters. Texture rectangles are screen-aligned rectangles whose coordinates are defined directly in screen space.

Example 15-1 Texture Rectangle Command

```
gsDPTextureRectangle(xl, yl, xh, yh, tile, s, t, dsdx, dtdy)
```

Texture coordinates are defined by specifying the start point S and T coordinates at the top left corner of the rectangle and the step in S per pixel in X and the step in T per pixel in Y. Example 15-2 shows a rectangle 100 pixels wide by 100 pixels high drawn at screen coordinates (100,100). The texture coordinates at the top left corner of the rectangle are (0,0). The texture steps 1 texel per pixel in both the S and T directions. This example assumes that a texture has been previously loaded (see "Texture Loading" on page 248).

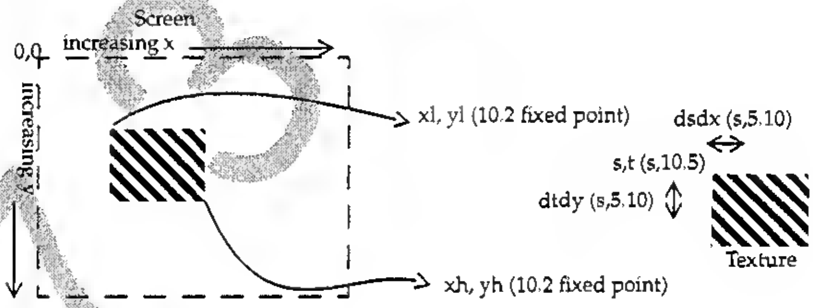
Example 15-2 Texture Rectangle Example

```
gsDPSetTexturePersp(G_TP_NONE),
gsDPTextureRectangle(100<<2, 100<<2, 200<<2, 200<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10),
```

Caution: The perspective divide of the texture coordinates in the RDP must be disabled using the *gsDPSetTexturePersp*() command when rendering texture rectangles.

Texture rectangles are two-dimensional (2D)-- they may be translated in X and Y, but not rotated. Texture rectangles may be z-buffered in a limited way, as described in "Z-Buffering Texture Rectangles" on page 299. Even though they are simple and limited to two dimensions, texture rectangles are useful both in 2-D sprite games as well as for 2-D effects in 3-D games. This chapter will explain some of the details associated with the texture rectangle primitive and provide some simple examples for new Nintendo-64 programmers. Some of the information found in this chapter may also be found in other chapters but is repeated here for completeness.

Figure 15-1 Texture Rectangle Definition

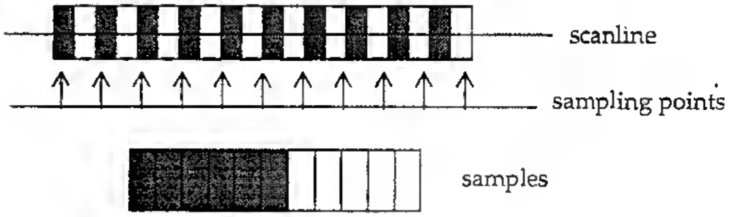

Screen
0,0
increasing x
increasing y
xl, yl (10.2 fixed point)
dsdx (s,5.10)
s,t (s,10.5)
dtdy (s,5.10)
Texture
xh, yh (10.2 fixed point)

Sampling Overview

A texture is an array of values, where each value is a set of numbers (*components*) describing the attributes of a texture element, or *texel*. For the Nintendo 64, the numbers representing a texel are fixed-point. The number of components per pixel and the number of bits per component is variable. "Color Index Frame Buffer" on page 298 describes the possible formats for texels.

When displaying a texture on the screen of a display, we must perform a mapping from the texture space to the display image space. In the case of texture rectangles, where the geometric operations are limited to scaling and translation, the main problem is how to sample and filter the source texture so that it is faithfully produced on the display. Figure 15-2 is one example of aliasing artifacts that can effect image quality. In this example, 10 black bars are separated by 10 white bars with even spacing. The bars cover a width of 11 pixels on the screen. Because we are sampling at a lower frequency than the texture, our output image is *aliased*. Aliasing artifacts are caused by high-frequency information that is insufficiently sampled appearing as low-frequency information. Furthermore, if the beginning sample point is moved slightly, the sampled image can shift dramatically. During animations this causes the displayed image to *scintillate* or flash. Nyquist's Law indicates that the sampling frequency should be greater than twice the highest frequency component in the texture to avoid aliasing artifacts.

Figure 15-2 Aliasing in a Sampled Image



Point Sampling

Point sampling in the Nintendo 64 means that we assume that each texel maps to one pixel on the display, and we ignore any fractional overlap

115

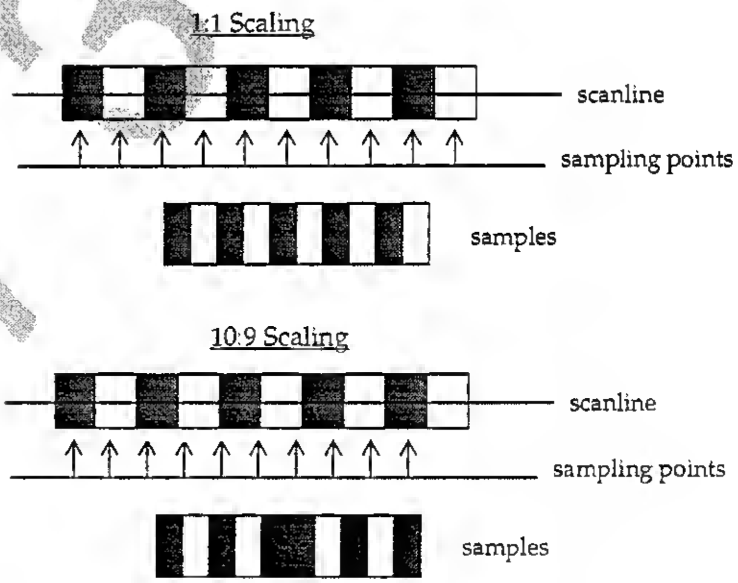
between texels and pixels. Example 15-3 shows how to enable point sampling.

Example 15-3 Enable Point Sampling

```
gsDPSetTextureFilter(G_TF_POINT)
```

Point sampling works well for mapping a rectangular texture to a screen-aligned rectangle of the same size on the display. Problems occur if the sampling ratio is not 1:1, however, as shown in Figure 15-3. In the first case, we display 10 texels using 10 pixels. In the second case, we scale the image slightly by displaying 9 texels on 10 pixels. This results in the middle pixel having the same color as the previous bar. In general, point sampled images should be scaled by an integer power of two to avoid this problem. To achieve other scalings, it is necessary to use bilinear filtering.

Figure 15-3 Point Sampling Scaling Problem



Example 15-4 demonstrates 3 texture rectangles with the texture scaled by 1, 2, and 4 respectively:

Example 15-4 Scaled, Point Sampled Textures

1157389

```
gsDPSetTextureFilter(G_TF_POINT),
gsDPTextureRectangle(50<<2, 50<<2, 150<<2, 150<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10),
gsDPTextureRectangle(60<<2, 60<<2, 160<<2, 160<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<9, 1<<9),
gsDPTextureRectangle(70<<2, 70<<2, 170<<2, 170<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<8, 1<<8),
```

Point sampling also implies that animated sprites will have to move in one-pixel increments. Even though the rectangle can be positioned with 2 bits of subpixel precision, and the texture can be offset to 5 bits of fractional precision, the point sampling only looks at the integer coordinate and so will not change until there is at least a one pixel change in position. Bilinear filtering allows for smoother motion of sprites.

Bilinear Filtering

Instead of selecting a single texel for a given pixel, as in point sampling, bilinear filtering selects four texels surrounding the sample point and intepolates these points using fractional position information to determine the pixel color. Example 15-5 shows how to enable texture filtering.

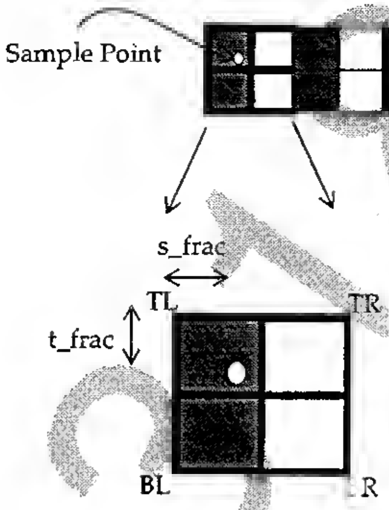
Example 15-5 Enable Bilinear Filtering

```
gsDPSetTextureFilter(G_TF_BILERP)
```

1157189

An example of bilinear filtering is shown in Figure 15-4.

Figure 15-4 Bilinear Filtering

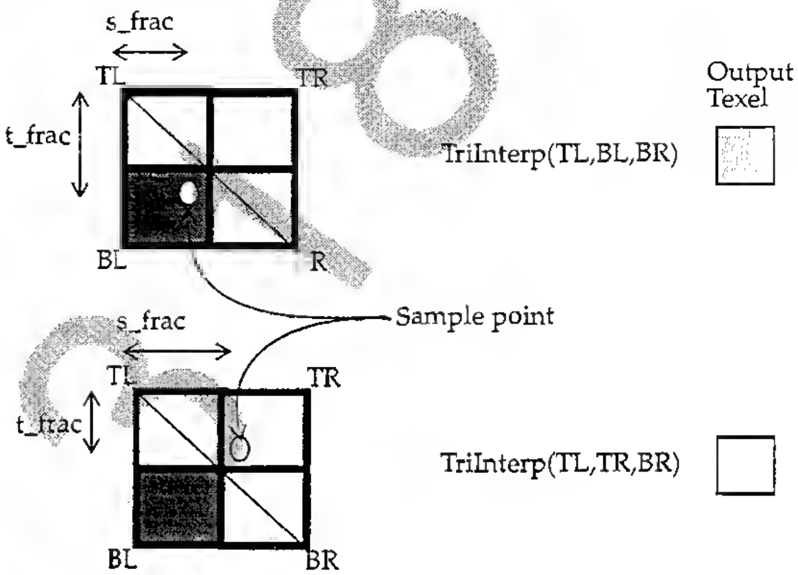


top =TL + s_frac(TR-TL)
bot =BL + s_frac(BR-BL)

texel =top + t_frac(bot-top)

In the Nintendo-64, rather than doing a full bilinear interpolation using all four samples, a triangular interpolation is performed that uses only three points. The texture filter selects which three points to use depending on where the sample point lies inside the 2x2 grid of texels. In certain cases, the triangular filter can cause small anomalies. These cases occur when there are drastic intensity changes from one texel to another in the texture as shown in Figure 15-5. In this example, if the sampling point moves slightly from one side of the diagonal to the other, the resulting color changes abruptly. In general, it is best to prefilter an image so that these sharp texture edges at least a slight intensity ramp.

Figure 15-5 Triangular Filtering


s_frac
TL
TR
t_frac
BL
R
Output
Texel
TriInterp(TL,BL,BR)
Sample point
s_frac
TL
TR
t_frac
BL
BR
TriInterp(TL,TR,BR)

With bilinear filtering, it is possible to scale a texture without the problems of point sampling. Example 15-6 shows a texture rectangle with the texture scaled by 1.5 in S and T:

Example 15-6 Scaled, Bilerped Textures

```
gsDPSetTextureFilter(G_TF_BILERP),
gsDPTextureRectangle(50<<2, 50<<2, 150<<2, 150<<2,
    G_TX_RENDERTILE,
    0, 0,
    3<<9, 3<<9),
```

Smooth scrolling of texture rectangles is discussed in "Smooth Scrolling" on page 286.

Average mode for 1:1 Ratio Sampling

There is a special case in which the texture filter can perform an exact average using all four texels. This case occurs when the sample point lies exactly in the center, i.e. s_frac = t_frac = 0.5. To enable the average mode use the command:

Example 15-7 Enable Average Filtering

```
gsDPSetTextureFilter(G_TF_AVERAGE)
```

In order to force the sample point to be in the middle of the texel, set the start point to 0.5 and then step by 1 texel per pixel. Example 15-8 demonstrates this:

Example 15-8 Averaging Textures

```
gsDPSetTextureFilter(G_TF_AVERAGE),
gsDPTextureRectangle(50<<2, 50<<2, 150<<2, 150<<2,
    G_TX_RENDERTILE,
    1<<4, 1<<4,
    1<<10, 1<<10),
```

Copy

Copy mode is a special pipeline mode that allows fast image copies to the framebuffer. Copy mode can be enabled as shown in

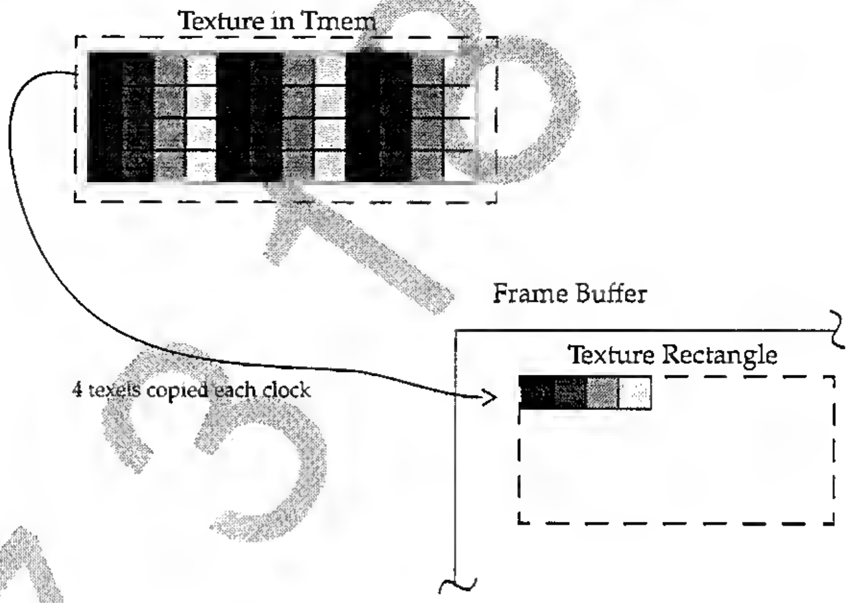
Example 15-9 Enable Copy Mode

```
gsDPSetCycleType(G_CYC_COPY)
```

11573189

In copy mode, four horizontally adjacent texels are copied per clock as shown in Figure 15-6.

Figure 15-6 Copy Mode



In copy mode, since four texels are copied each clock, the step in S per clock must be set to four. Example 15-10 shows a texture rectangle using copy mode.

Example 15-10 Copy Mode Texture Rectangle

```
gsDPSetCycleType(G_CYC_COPY),
gsDPTextureRectangle(50<<2, 50<<2, 150<<2, 150<<2,
    G_TX_RENDERTILE,
    0, 0,
    4<<10, 1<<10),
```

Since copy mode bypasses most of the RDP pipeline, the filter settings are not used. However, it is still necessary to disable perspective correction as shown in Example 15-2. Also, copy mode is not valid for all texture types, see "Copy" on page 259.

115731809

It is possible to scale textures in copy mode in the T(Y) direction only. Note that in this case, the rules for point sampled scaling apply, only integer power of two scalings.

In copy mode, textures are copied directly to memory, so there is no opportunity for color combiner operations, filtering, transparency, etc. Copying is a write-only operation so transparency using the normal blending hardware is impossible. However, you can achieve 'cutout' and 'dithered' types of transparency using the alpha compare logic, see "Alpha Compare Calculation" on page 315.

1157319

1157389

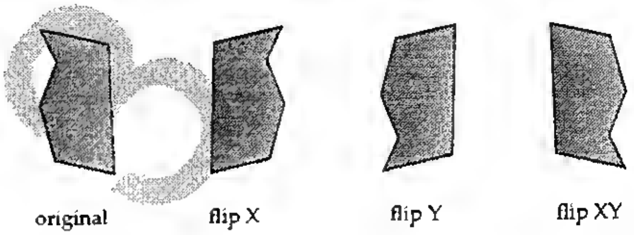
Simple Texture Effects

This section describes some 'sprite'-type effects that are commonly useful for texture rectangles. This is intended to be a starting point for programmers, not a complete list. Undoubtedly, clever programmers will find the hardware allows many other effects.

Flip

Flip means to rotate an image 180 degrees around the X or Y axis or both as shown in Figure 15-7.

Figure 15-7 Flipping Texture Rectangles



If the texture map to be flipped has a size that is a power of two in the direction of the flip, then you can use the mirror_enable ("Mirror Enable S,T" on page 222) bit in the tile descriptor to perform the flip. For example, suppose we have loaded a 32x32 16-bit RGBA texture into Tmem. To flip the texture in X we can use the code in Example 15-11.

Example 15-11 Flip a Texture in X

```
gsDPSetTile(G_IM_FMT_RGBA, G_IM_SIZ_16b, 8, 0,
    G_TX_RENDERTILE,0,
    G_TX_MIRROR, 5, G_TX_NOLOD, /* s */
    G_TX_NOMIRROR, 5, G_TX_NOLOD), /* t */
gsDPTextureRectangle(50<<2, 50<<2, 150<<2, 150<<2,
    G_TX_RENDERTILE,
    32<<5, 0, /* start s on mirror boundary */
    1<<10, 1<<10),
```

Note that the S start point is 32. Since the texture will be mirrored when the S coordinate is between 32 and 63 if the mirror enable bit in the tile is set, we get the effect of a flipped texture. If the mirror bit is disabled, the texture will remain unflipped.

For textures that are not power of two sizes, we must use another approach for flipping the textures. Suppose we have loaded a 48x42 16-bit RGBA texture in Tmem and would like to flip the texture in T. The code in Example 15-12 would accomplish this.

Example 15-12 Flip a Texture in Y (non power-of-two size)

```
gsDPTextureRectangle(50<<2, 50<<2, 98<<2, 92<<2,
    G_TX_RENDERTILE,
    0, 41<<5, /* start t at bottom of texture */
    1<<10, ((-1)<<10)&0xffff), /* step from bottom to top of
    texture*/
```

Note that we change the texture T coordinate to start at the bottom of the texture and change the increment in T so that we step from the bottom of the texture to the top, thus flipping the texture in Y.

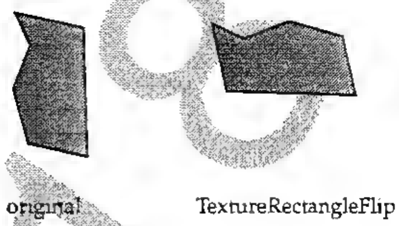
There is also a variation of the texture rectangle called *g*DPTextureRectangleFlip()* that swaps the S and T coordinates in hardware. If we had a display list as in Example 15-13

Example 15-13 TextureRectangleFlip command

```
gsDPTextureRectangleFlip(50<<2, 50<<2, 98<<2, 92<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10)
```

we would get an resulting image as shown in Figure 15-8.

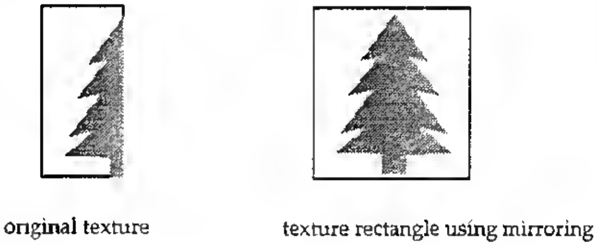
Figure 15-8 TextureRectangleFlip Command


original
TextureRectangleFlip

Mirror

Mirroring is also useful for data compression in cases where the texture has axial symmetry. For example, a tree could be created with half of a tree texture that was mirrored in X as shown in Figure 15-9.

Figure 15-9 Mirrored Tree


original texture
texture rectangle using mirroring

As mentioned before, to use hardware mirroring, the texture must be a power of two size in the direction to be mirrored. Suppose the tree texture above is a 16x40 16-bit RGBA texture. Example 15-14 will render the mirrored tree as shown in Figure 15-9.

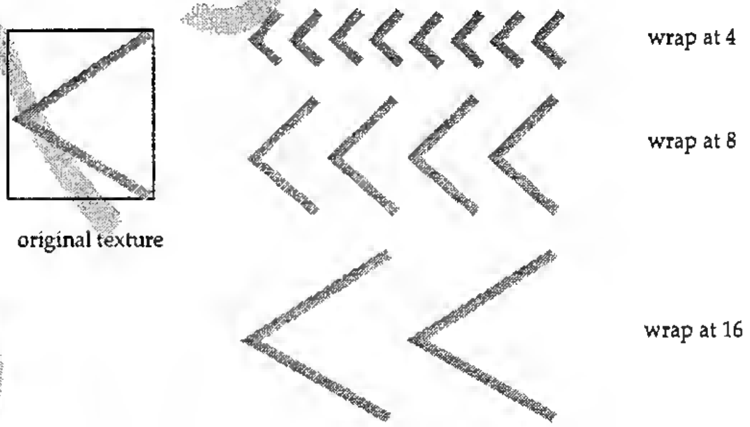
Example 15-14 Mirrored Tree

```
gsDPLoadTextureTile(tree, G_IM_FMT_RGBA, G_IM_SIZ_16b,
    16, 40,
    0, 0, 15, 39,
        0,
        G_TX_MIRROR, G_TX_CLAMP,
        4, G_TX_NOMASK,
        G_TX_NOLOD, G_TX_NOLOD),
gsDPTextureRectangle(50<<2, 50<<2, 82<<2, 90<<2,
        G_TX_RENDERTILE,
        0, 0,
        1<<10, 1<<10),
```

Wrap

Wrapping allows a small texture to fill a larger rectangle by repeating the texture over and over. In the Nintendo-64, wrapping is enabled if the mask (see "Mask S,T" on page 223) in the tile descriptor is non-zero and the clamp bit (see "Clamp S,T" on page 224) in the tile descriptor is not set for the coordinate in question. The mask determines which power of two the wrap occurs on. Figure 15-10 shows the results for various wrap boundaries using a single texture. Wrapping can be used in copy mode except for

Figure 15-10 Wrapping on Several Boundaries of the Same Texture



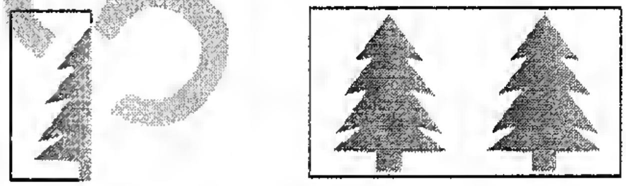
Wrapping can also be used in conjuction with mirroring. Suppose we wanted to wrap the mirrored tree shown in Figure 15-9. This could be done using the code in Example 15-15.

Example 15-15 Wrapped and Mirrored Tree

```
gsDPLoadTextureTile(tree, G_IM_FMT_RGBA, G_IM_SIZ_16b,
    16, 40,
    0, 0, 15, 39,
    0,
    G_TX_MIRROR | G_TX_WRAP, G_TX_CLAMP,
    4, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
gsDPTextureRectangle(50<<2, 50<<2, 114<<2, 90<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10),
```

Note that the G_TX_WRAP above is really unnecessary because wrapping is implicit as we have a non-zero mask value and are not clamping. It is included just for documentation purposes. The resulting image would look like Figure 15-11.

Figure 15-11 Wrapped and Mirrored Tree



original texture

texture rectangle using wrapping and mirroring

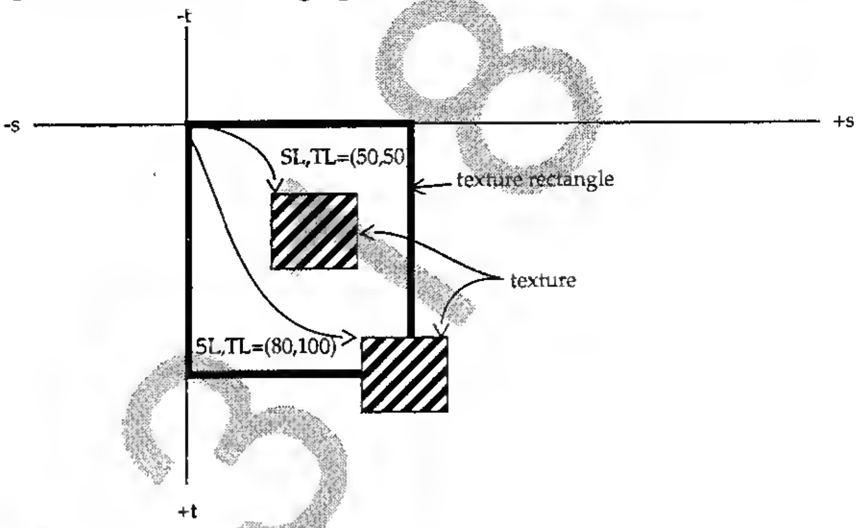
Sliding Textures

It is easy to slide a texture relative to the rectangle primitive by the changing the tile descriptor values of SL and TL (see "SL,TL" on page 224). Using the

1157389

tile descriptor allows the texture coordinates to be statically defined. The effect of changing SL, TL is shown in Figure 15-12.

Figure 15-12 Effect of Changing SL, TL



Suppose we have a 32x32 4-bit I texture loaded in Tmem. In Example 15-16, two rectangles are rendered with the texture placed in different positions using SL and TL.

Example 15-16 Sliding Texture Using SL, TL

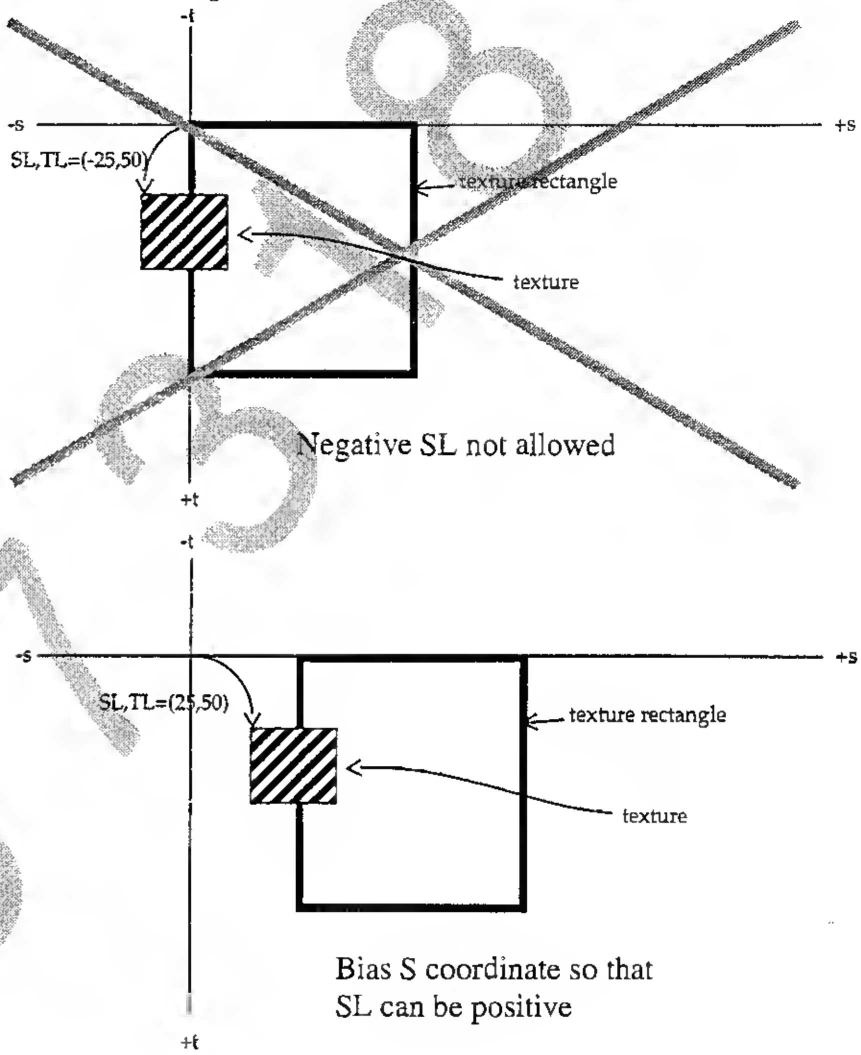
```
gsDPSetTileSize(G_TX_RENDERTILE, 50, 50, 82, 82),
gsDPTextureRectangle(50<<2, 50<<2, 82<<2, 82<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10),
gsDPSetTileSize(G_TX_RENDERTILE, 80, 100, 112, 132),
gsDPTextureRectangle(100<<2, 100<<2, 132<<2, 132<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10),
```

Note that SH and TH are only used when clamping. Because SL and TL are unsigned, the texture rectangle coordinates must be offset to allow sliding

1157389

above the top edge or to the left of the left edge of the rectangle. This is shown in Figure 15-13 and Example 15-17.

Figure 15-13 Biasing Texture Coordinates for Positive SL, TL


-t
-s
+s
SL,TL=(-25,50)
texture rectangle
texture
Negative SL not allowed
+t
-t
-s
+s
SL,TL=(25,50)
texture rectangle
texture
Bias S coordinate so that
SL can be positive
+t

Example 15-17 Biased Coordinates for Positive SL

```
gsDPSetTileSize(G_TX_RENDERTILE, 25, 50, 57, 82),
gsDPTextureRectangle(50<<2, 50<<2, 82<<2, 82<<2,
```

11573189

```
    G_TX_RENDERTILE,
    50<<5, 0,
    1<<10, 1<<10),
```

Smooth Scrolling

Scrolling involves positioning texture rectangles on the screen and also positioning the texture within the rectangle. The rectangle geometry can be positioned with 2 bits of fractional precision in X and Y. The texture coordinates can be specified with 5 bits of fractional precision in S and T. To get the smoothest scrolling, you can use the S and T start point as the fractional part and the rectangle's X and Y position for the integer part. So effectively, you are sliding the texture to achieve fractional displacements. Example 15-18 shows how such positioning could be achieved. Keep in mind that a border area around the texture must be present so that the texture doesn't clamp when it slides off the rectangle.

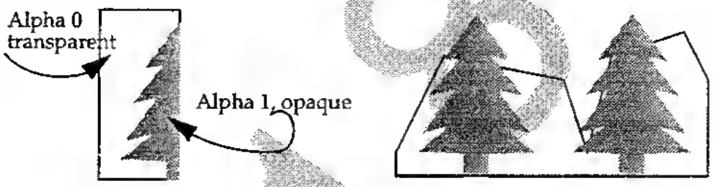
Example 15-18 Accurate Positioning Using S and T

```
float xpos = 10.375, ypos = 19.432;
int xi, xf, yi, yf;

xi = (int) xpos;
yi = (int) ypos;
xf = 32 - 32 * (xpos - xi);
yf = 32 - 32 * (ypos - yi);
gDPTextureRectangle(glistp++,
    xi<<2, yi<<2, (xi+32)<<2, (yi+32)<<2,
    G_TX_RENDERTILE,
    xf, yf,
    1<<10, 1<<10);
```

Billboards

Billboards are textures that define complex outlines by using texture transparency. For example, rather than creating a tree using polygons, you can use an image of a tree, with the portion of the image outside the tree having an alpha of 0 (transparent) and the interior of the tree having an alpha of 1 (opaque). This is shown graphically in Figure 15-14. This technique allows complex scenes to be built by compositing simple images together.

Figure 15-14 Texture Billboard



original texture texture rectangle using wrapping and mirroring

It is important to consider the antialiasing of the edges created by the texture's alpha pattern. If only 1 bit of alpha is used, then the pixel is either written or not. If more bits of alpha are used to create a smoother transition from opaque to transparent the edges will be blended with the background. Billboards should be rendered after all opaque background objects have been rendered. There are several texel formats that allow multiple bits of alpha (see "Color Index Frame Buffer" on page 298) and ways of combining different types (see "Combining Types" on page 290). To render this type of antialiased texture billboard, you must be in one or two cycle mode and you should use the render mode G_RM_AA_TEX_EDGE. See "Texture Edge Mode, TEX_EDGE" on page 332 for further details.

Texture billboards can also be rendered in a write-only fashion but this also implies no antialiasing of the texture edge. This mode is called 'alpha compare' and basically thresholds the texel alpha with a register alpha value or a random alpha source to generate a write enable for the pixel. See "Alpha Compare Calculation" on page 315 for more details.

Cloud (CLD) Render Mode

Cloud render mode is intended for rendering texture billboards that are not opaque, i.e. smoke clouds, explosions, etc. These are special cases because care must be taken not to disturb the antialiased edges of things behind the transparent cloud, because these edges will be seen through the cloud.

1153109

Texture Types

Intensity (I) Textures

Intensity textures are useful because they are quite compact and should be used in cases where a large number of colors is not necessary. For example, a 4-bit I texture can be as large as 128x64 texels. Normally, the user would like the primitive to have some specific color, and the I texture should modulate that color. For example, to create a tree you could use two I textures, one for the brown trunk and one for the green treetop. You can use one of the many register colors in the color combiner to define the primitive color. In Example 15-19 we use primitive color to define the colors of the trunk and treetop.

Example 15-19 Intensity Texture Modulating Primitive Color

```
gsDPSetCombineMode(G_CC_MODULATEI_PRIM, G_CC_MODULATEI_PRIM),
gsDPSetPrimColor(0, 0, 205, 51, 51, 255), /* brown */
gsDPLoadTextureTile_4b(trunk, G_IM_FMT_I, 16, 40,
    0, 0, 15, 39,
    0,
    G_TX_MIRROR, G_TX_CLAMP,
    4, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
gsDPTextureRectangle(50<<2, 100<<2, 82<<2, 140<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10),
gsDPSetPrimColor(0, 0, 0, 139, 0, 255), /* green */
gsDPLoadTextureTile_4b(treetop, G_IM_FMT_I,32, 32,
    0, 0, 15, 39,
    0,
    G_TX_MIRROR, G_TX_CLAMP,
    5, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
gsDPTextureRectangle(44<<2, 68<<2, 108<<2, 100<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10),
```

By interpolating between two different colors using the intensity as the parameter, it is possible to achieve two-color textures. The combine mode

G_CC_BLENDPEDECALA interpolates between primitive color and environment color using an I texture. For this combine mode, when the texture is 0 the pixel will be environment color, when the texel is all ones, the pixel will be primitive color. Example 15-20 assumes an I texture has already been loaded into Tmem.

Example 15-20 Two-Color Texture

```
gsDPSetCombineMode(G_CC_BLENDPEDECALA, G_CC_BLENDPEDECALA),
gsDPSetPrimColor(0, 0, 205, 51, 51, 255), /* brown */
gsDPSetEnvColor(0, 0, 0, 200, 0, 255), /* green */
gsDPTextureRectangle(80<<2, 100<<2, 82<<2, 140<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10),
```

Since for intensity textures the texel value is also copied onto the alpha channel, you can achieve transparency using an intensity texture. For example, if you define a 4-bit texture of some text to have an intensity of 0xf for the characters and a value of 0 elsewhere, and then render using the combine mode G_CC_BLENDPEDECALA and the render mode G_RM_TEX_EDGE, the text will have the primitive color and be transparent elsewhere. Note that if the edges of the text are filtered to give smooth edges, then the text will have an intensity ramp at the edges. If you use an antialiased render mode, such as G_RM_AA_TEX_EDGE, then the text will look smoother than if a 1-bit alpha texture like 4-bit IA or 16-bit RGBA were used.

Intensity Alpha (IA) Textures

This texture type defines an intensity (I) channel and a separate alpha channel (A). This type is convenient where the transparency of the texture must be defined separately from the intensity. The sizes include 4-bit (3 bits of I and 1 bit of A), 8-bit (4 bits of I and 4 bits of A), 16-bit (8 bits of I and 8 bits of A). Keep in mind when using 1-bit alphas that the pixel will be either written or not, depending on the alpha bit. Therefore, the transparency channel is not antialiased (the texture filter cannot 'create data' to smooth the edge). Scaling a 1-bit alpha texture can result in blocky-looking outlines.

Color (RGBA) Textures

There are two sizes of RGBA textures: 16-bit (5 bits R, 5 bits G, 5 bits B, 1 bit A), and 32-bit (8 bits R, 8 bits G, 8 bits B, 8 bits A). While 16-bit RGBA textures are popular because they are easy to create and model with, they have the disadvantage of only a 1-bit alpha channel. This can be overcome in certain cases, as discussed in "Combining Types" on page 290.

Color Index (CI) Textures

Color index textures come in two sizes, 8-bit and 4-bit. When using color index textures only half the Tmem is used for textures (2KBytes). The other half is used to store the lookup table (TLUT) that converts the index texel into either 16-bit RGBA or 16-bit IA types. It is also possible to copy 8-bit CI textures directly to an 8-bit framebuffer as discussed in "Color Index Frame Buffer" on page 298.

4-bit CI textures must select one of 16 possible palettes. Each palette has 16 entries. The *g*DPLoadTLUT_pal16* can be used to load an individual palette. The palette to use is defined in the tile descriptor (normally you would define the palette in the *g*DPLoadTexture** command), so different tiles can select different palettes.

You can use a 4-bit CI texture to provide more alpha bits than is possible with the 4-bit IA type, because the TLUT can hold 16-bit IA values. Therefore, you could look up 16 levels of alpha with a 4-bit CI sprite as compared to 1 level for a 4-bit IA sprite.

Combining Types

As mentioned previously, 16-bit RGBA textures have only a 1-bit alpha channel. If you want to have a smoothly antialiased texture edge using the 16-bit RGBA type, you must combine two types of texture. Example 15-21 shows how a separate alpha texture with a 4-bit I type is combined with a 16-bit RGBA type to get smoother edges on a sprite.

Example 15-21 Interpolate Between Two Tiles

```
#define MULTIBIT_ALPHA  0, 0, 0, TEXEL0, 0, 0, 0, TEXEL1

gsDPSetCyleType(G_CYC_2CYCLE),
gsDPSetTextureLOD(G_TL_TILE),
gsDPSetCombineMode(MULTIBIT_ALPHA, G_CC_PASS2),
gsDPSetRenderMode(G_RM_AA_TEX_EDGE, G_RM_AA_TEX_EDGE2),
/* load color part of texture */
gsDPLoadMultiTile(color,
    0, /* Tmem address in 64-bit words */
    G_TX_RENDERTILE, /* tile */
    G_IM_FMT_RGBA, G_IM_SIZ_16b,
    32, 32,
    0, 0, 31, 31,
    0,
    G_TX_NOMIRROR, G_TX_NOMIRROR,
    G_TX_NOMASK, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
/* load alpha part of texture */
gsDPLoadMultiTile_4b(alpha,
    256, /* Tmem address in 64-bit words */
    G_TX_RENDERTILE+1, /* tile */
    G_IM_FMT_I,
    32, 32,
    0, 0, 31, 31,
    0,
    G_TX_NOMIRROR, G_TX_NOMIRROR,
    G_TX_NOMASK, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
gsDPTextureRectangle(glistp++,
    50<<2, 50<<2, 82<<2, 82<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10);
```

The idea here is that in two-cycle mode we get two texel values, one from the 16-bit RGBA texture and one from the 4-bit I texture. In the color combiner, we program the alpha combiner to use the 4-bit I texture (the 1-bit A of the RGBA texture is not used). In the color combiner, we select the RGB texture as the color source. Since we are using both cycles for this trick, it is not possible to use mipmapping or other two-cycle modes simultaneously. Note that you could have used an 8-bit I texture for the alpha channel if you needed more alpha resolution.

11573189

Multi-Tile Effects

There are eight tile descriptors available in the tile memory of the RDP. These tile descriptors contain information about the type and size of tiles and where these tiles are located in Tmem. In two-cycle mode, texture from two tiles is available for each pixel. Many effects are possible by manipulation of tile descriptors and combining of the textured pixels.

In the *g*DPLoadTexture** commands, a simple two-tile system is used for loading and rendering. In this system, the G_TX_LOADTILE is used for loading a tile starting at Tmem address 0 and the tile descriptor G_TX_RENDERTILE is set up for rendering the tile. This is a double-buffering scheme which avoids having to insert tile sync commands in the load macro. Notice that since each tile is loaded at Tmem address 0 and the G_TX_RENDERTILE is always used for rendering, we cannot use these macro for loading multiple tiles into Tmem.

In order to allow the user to manage Tmem for multi-tile effects, the load macros *g*DPLoadMultiTile* and *g*DPLoadMultiBlock* were created. These macros allow the user to specify the Tmem address of the tile and the tile descriptor number to use when rendering this tile.

Simple Morph

One simple use of two tiles is to linearly interpolate, using a parameter to indicate the blend amount, between the tiles. A register value in the color combiner, such as primitive alpha, can be used as the 'slider' to blend between the two textures as shown in Example 15-22. Notice that we define our own color combine mode to achieve this effect, since gbi.h didn't have the mode we needed.

Example 15-22 Interpolate Between Two Tiles

```
#define MY_MORPH TEXEL1, TEXEL0, PRIMITIVE_ALPHA, TEXEL0, \
    TEXEL1, TEXEL0, PRIMITIVE, TEXEL0

gsDPSetCyleType(G_CYC_2CYCLE),
gsDPSetTextureLOD(G_TL_TILE),
gsDPSetPrimColor(0, 0, 0, 0, 0, 128), /* 0.5 blend */
gsDPSetCombineMode(MY_MORPH, G_CC_PASS2),
gsDPLoadMultiTile(face0,
    0, /* Tmem address in 64-bit words */
    G_TX_RENDERTILE, /* tile */
    G_IM_FMT_RGBA, G_IM_SIZ_16b,
    32, 32,
    0, 0, 31, 31,
    0,
    G_TX_NOMIRROR, G_TX_NOMIRROR,
    G_TX_NOMASK, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
gsDPLoadMultiTile(face1,
    256, /* Tmem address in 64-bit words */
    G_TX_RENDERTILE+1, /* tile */
    G_IM_FMT_RGBA, G_IM_SIZ_16b,
    32, 32,
    0, 0, 31, 31,
    0,
    G_TX_NOMIRROR, G_TX_NOMIRROR,
    G_TX_NOMASK, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
gsDPTextureRectangle(glistp++,
    50<<2, 50<<2, 82<<2, 82<<2,
    G_TX_RENDERTILE,
    0, 0,
    1<<10, 1<<10);
```

By making the primitive alpha an animation variable, a simple 'morph' effect can be achieved.

Smoothing Flip-Book Animations

Often sprite animations are a sequence of key frames which are selected at the appropriate time by some animation variable. The linear interpolation between two images as described in "Simple Morph" above can be used to smoothly transition between two key frames. Imagine a series of *n* images in an animation selected using an animation variable *frame*. The integer part of *frame* is called *frame_i* and the fractional part is called *frame_f*. An algorithm for smoothing the sequence is described in Example 15-23.

Example 15-23 Smoothing an Animation Sequence

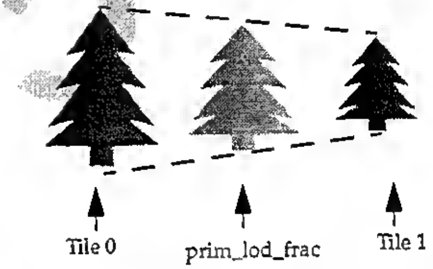
```
Load tiles frame_i and frame_i+1 into Tmem
Set primitive alpha = 256 * frame_f
Render the rectangle using MY_MORPH combiner mode
```

The frames do not necessarily have to be related in time. For example, you could interpolate between different flame images that are randomly selected to create a fire effect.

Shrinking Sprites

In the previous discussion of scaling in "Bilinear Filtering" on page 273 we only discussed scaling a sprite to a larger size since scaling it smaller would result in aliasing effects. It is possible to effectively shrink an image by interpolating between two tiles, one of which is a half the size of the other tile. This is shown in Figure 15-15. Prim_lod_frac is a register in the color combiner that can be used to indicate the fractional distance between the two 'levels-of-detail' of the sprite. Note that there is no special reason we used this register as the interpolation parameter, other than it's name suggests this use.

Figure 15-15 Shrinking a Sprite



One of the tile descriptor parameters is the *shift* (see "Shift S,T" on page 223) that describes how many places to bitwise shift the tile coordinates for the primitive. This implies that one tile's size is related to the other's by some integer shift, but the tiles don't necessarily have to be power of two sizes. Example 15-24 shows the code to create a sprite that is 0.75 the size of the larger image. The user must scale the size of the rectangle primitive by the desired amount as well.

Example 15-24 Shrinking a Sprite

```
#define MY_LOD TEXEL1, TEXEL0, PRIM_LOD_FRAC, TEXEL0, \
    TEXEL1, TEXEL0, PRIM_LOD_FRAC, TEXEL0

gsDPSetCyleType(G_CYC_2CYCLE),
gsDPSetTextureLOD(G_TL_TILE),
```

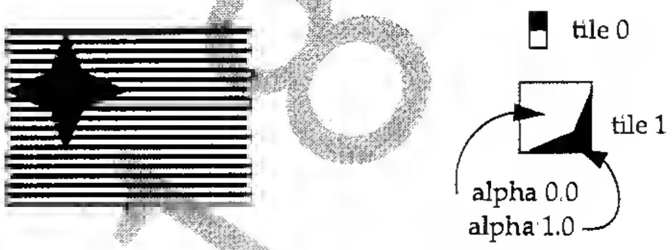
1157389

```
gsDPSetPrimColor(0, 128, 0, 0, 0, 0), /* 0.5 lod_frac */
gsDPSetCombineMode(MY_LOD, G_CC_PASS2),
gsDPLoadMultiTile(face0,
    0, /* Tmem address in 64-bit words */
    G_TX_RENDERTILE, /* tile */
    G_IM_FMT_RGBA, G_IM_SIZ_16b,
    32, 32,
    0, 0, 31, 31,
    0,
    G_TX_NOMIRROR, G_TX_NOMIRROR,
    G_TX_NOMASK, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
gsDPLoadMultiTile(face1,
    256, /* Tmem address in 64-bit words */
    G_TX_RENDERTILE+1, /* tile */
    G_IM_FMT_RGBA, G_IM_SIZ_16b,
    16, 16,
    0, 0, 15, 15
    0,
    G_TX_NOMIRROR, G_TX_NOMIRROR,
    G_TX_NOMASK, G_TX_NOMASK,
    G_TX_NOLOD, G_TX_NOLOD),
gsDPTextureRectangle(glistp++,
    50<<2, 50<<2, 82<<2, 82<<2,
    G_TX_RENDERTILE,
    8<<5, 8<<5,
    1<<10, 1<<10);
```

Texture Decals

We can use the alpha of one tile to select between the texel color of two different tiles to create a texture decal. Figure 15-16 shows an example of a flag created using textures decals. The insignia of the flag has transparency around it's edges. After mirroring and wrapping once, the texture is clamped. In the color combiner, the texture alpha is used to interpolate between the flag stripes and the insignia. Where the alpha is zero, the stripes will show, where the alpha is one, the insignia will show.

Figure 15-16 Texture Decals

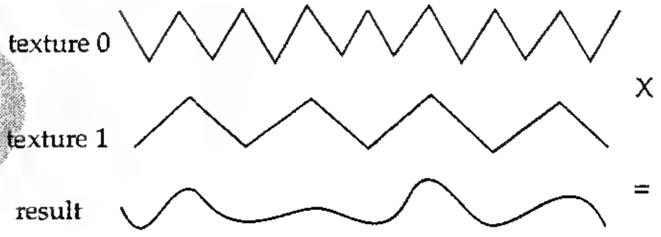


Need example code...

Interference Effects

Multiplying two textures together, especially while sliding the textures relative to each other can create interference patterns. For example, a horizontal stripe pattern multiplied by a vertical stripe pattern creates a set of bright spots at the intersection of the points. If the stripes are slid relative to each other, the points will move also. Multiplying can also be used to modulate one image with another. For example, Figure 15-17 shows a complex wave resulting from the modulation of two simple waves.

Figure 15-17 Modulation



1153189

Tiling Large Images

Sometimes it is desirable to render large textures, i.e. textures to large to fit entirely into Tmem. This can be accomplished via 'tiling' or breaking the large image up into smaller rectangular tiles that do fit into Tmem. These tiles are rendered onto primitives that form a mesh coincident with the texture tiling. The textured rectangle primitive is a useful primitive for tiling a background image in a sprite game, for instance. If you point sample the texture tile, it is only necessary to load the number of texels you wish to display. However, if you want to bilinearly filter the texture, you must load a border region of one texel around the tile so that the interpolation works correctly at the edges of the tile. See "Bilinear Filtering and Point Sampling" on page 236 for more information.

1157389

Color Index Frame Buffer

You might have noticed that one of the *color* image types that is available is the 8-bit I type. You can use this mode to render color index images into the framebuffer. Before displaying the 8-bit image, however, you must read the 8-bit image into Tmem and dereference into a 16-bit RGBA image. Note that the 8-bit frame buffer can share the same memory as the 16-bit frame buffer by placing the 8-bit buffer in the high half of the 16-bit buffer.This technique can give better performance than rendering directly to a 16-bit framebuffer because the memory accesses are more efficient. Also, the initial clear of the framebuffer is faster because the buffer is half the size.

There are, however, restrictions when using this technique. Since we are rendering an 8-bit CI image, you must texture map objects with 8-bit CI textures (but don't dereference yet) and use shade colors that fit into your palette. You cannot filter the textures since the texture values in the pipeline are indices. You also cannot blend with memory colors (unless your palette is laid out specifically to allow this), although you can achieve cut-out type transparency. Antialiasing is also not available for this framebuffer type, because no coverage is stored.

These restrictions sound severe, but may be practical for some sprite games, especially those that use sort priority and can render totally in copy mode. In copy mode (and 1 or 2-cycle mode) you can get cut-out transparency by using the alpha compare logic and reserving an index (0 is a good choice) that indicates transparency. If the index 0 means transparent, then setting the blend alpha to 1 and enabling alpha compare (G_AC_THRESHOLD) would allow all pixel with any index greater than or equal to 1 to be written to the framebuffer but pixels with index 0 would not be written.

11571569

Z-Buffering Texture Rectangles

Normally, sprites are rendered in a Z sorted list and rendered from back to front. The Z of each sprite must be maintained by the application and the application must do the sort each frame. Another technique is to use the z-buffer to determine priority.

Primitive Z

The texture rectangle has no Z value associated with it directly, however you can use the primitive Z register (*g*DPSetPrimDepth()*). To force the z-buffer logic to use primitive Z rather than pixel Z, you must use the following command:

```
gsDPSetDepthSource(G_ZS_PRIM)
```

You must also use a RenderMode that enables z-buffering, such as G_RM_ZB_OPA_SURF. To z-buffer sprites, you would have to insert a g*DPSetPrimDepth() command before the rectangle command of each sprite. Because the primitive Z is explicitly buffered in the pipeline, it is not necessary to insert pipe sync commands before setting the register.

Note that z-buffering can only be used in 1 and 2-cycle mode. In copy and fill mode, you should use the RenderMode G_RM_NOOP to effectively disable z-buffering and put the pipeline logic in a safe state.

1151789

11573189

Chapter 16

Antialiasing and Blending

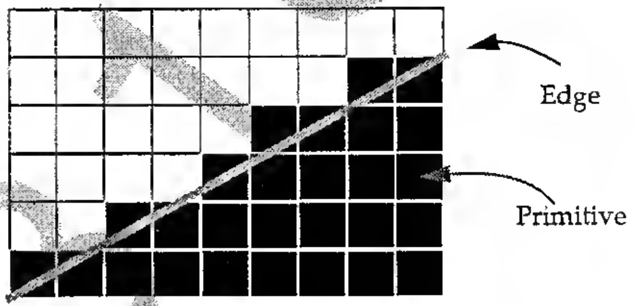
Aliasing is a signal-processing term describing sampling errors that occur when a continuous function containing sharp changes in intensity is approximated using discrete intensity values. *Antialiasing* is a method for minimizing these errors by using gradations in intensity of neighboring pixels at edges of primitives, rather than setting pixels to maximum or zero intensity only. There are many references on antialiasing as it applies to graphics. This chapter will discuss the method of antialiasing used by the Reality Co-Processor (RCP). In addition, we will discuss other uses of the *blender* hardware. The blender plays a key role in antialiasing, z-buffering, and transparency effects. After understanding the blender hardware, it may be possible for a user to come up with new effects by clever programming of the blender pipeline.

115189

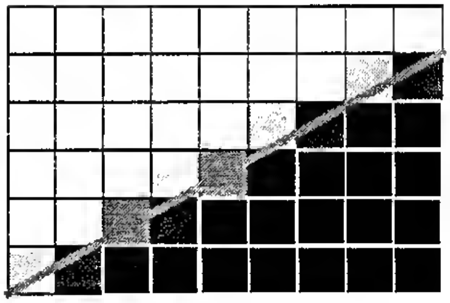
Antialiasing

Antialiasing is an algorithm that attempts to minimize sampling errors that occur when an edge of a primitive is displayed on a raster image. Visually, these errors cause the edge to be *stair-cased* or look *jaggy*. For scenes with moderate complexity and/or animation, these jaggies are the source of high-frequency noise, which is annoying and distracting to users.

Figure 16-1 Edge With and Without Antialiasing



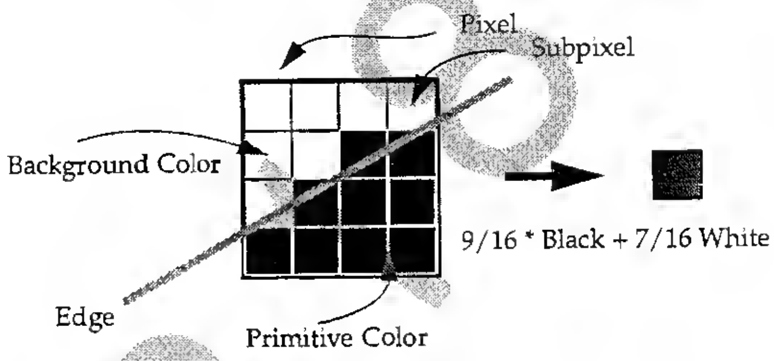
Aliased Edge



Antialiased Edge

In Figure 16-2, "Unweighted Area Sampling," on page 303, antialiasing is achieved by weighting the intensity of the pixel in proportion to the area of the pixel *covered* by the edge. In signal-processing terms, this is called *unweighted area sampling*.

Figure 16-2 Unweighted Area Sampling

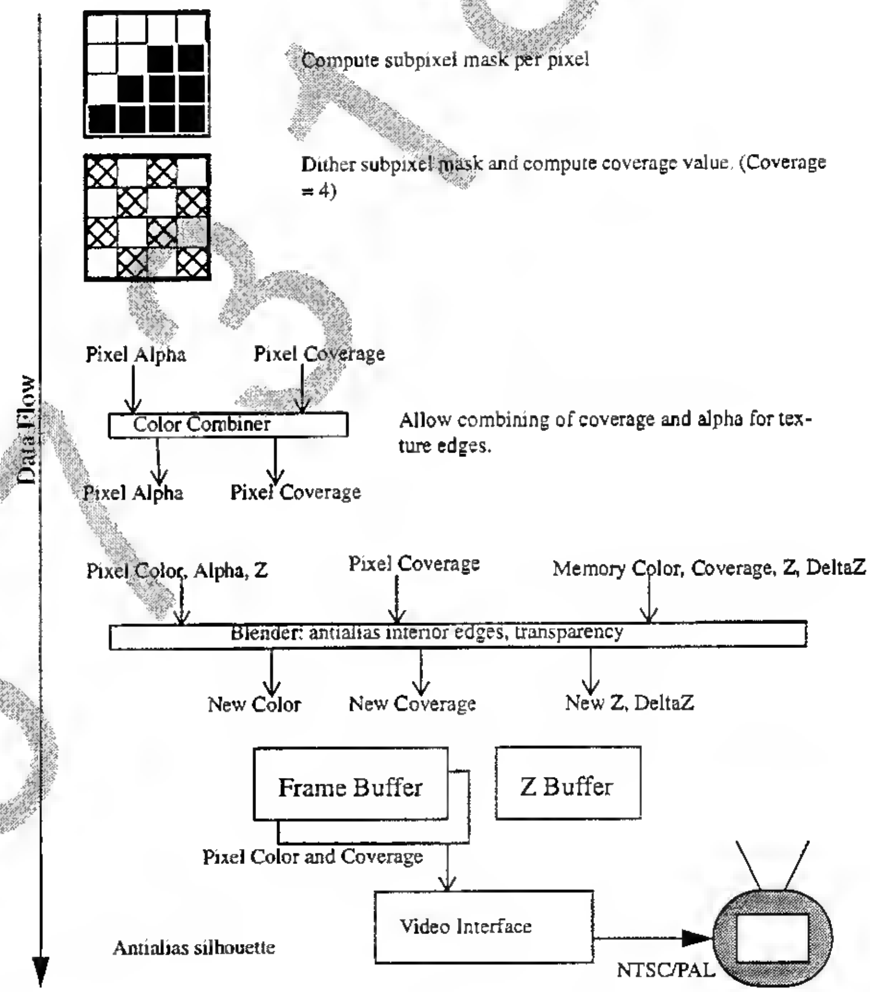


High-end graphics machines typically use an antialiasing technique known as *super-sampling*, in which the pixel is divided into a grid of sub-pixels. A color is computed for each subpixel and the subpixels that are covered by a primitive are averaged to produce the final pixel color. In the case where more than one primitive covers a pixel, each primitive's color is weighted by the number of subpixels it covers. Also, depth (Z) can be found for each subpixel which allows antialiased *interpenetrations* between primitives. While super-sampling is straightforward and effective, it is also expensive in terms of memory and memory bandwidth. For a 4x4 subpixel grid, 16 color and Z values must be stored for each pixel. In addition, to achieve required fill rates, each of these values must be accessed every clock.

Because the Nintendo 64 machine has very severe cost and memory requirements, a new and novel technique for antialiasing that avoided (as much as possible) the storage requirements of super-sampling but yet provided satisfactory antialiasing was needed. This method relies heavily on the notion that different *objects* have different antialiasing needs, and that the hardware can be simplified by requiring that different *RenderModes* are configured as appropriate for a particular object. As well, there are display-order restrictions for rendering certain types of objects. For example, transparent objects must be rendered after all the opaque objects. Finally, it was recognized that antialiasing of *silhouettes* could be done as a post process during video output. A data flow diagram of the analogizing algorithm is shown in Figure 16-3, "Antialiasing Data Flow," on page 304.

Note that this method requires, in addition to the pixel color and Z value, three bits of *coverage* and four bits of *deltaZ* per pixel, quite small when compared with super-sampling methods.

Figure 16-3 Antialiasing Data Flow


Data Flow
Compute subpixel mask per pixel
Dither subpixel mask and compute coverage value. (Coverage = 4)
Pixel Alpha
Pixel Coverage
Color Combiner
Allow combining of coverage and alpha for texture edges.
Pixel Alpha
Pixel Coverage
Pixel Color, Alpha, Z
Pixel Coverage
Memory Color, Coverage, Z, DeltaZ
Blender: antialias interior edges, transparency
New Color
New Coverage
New Z, DeltaZ
Frame Buffer
Z Buffer
Pixel Color and Coverage
Video Interface
Antialias silhouette
NTSC/PAL

1157318 9

The antialiasing data flow shows the most general case for z-buffered and antialiased primitives. Other techniques are possible. For example, if the database is sorted and rendered in back to front order, non-z-buffered antialiasing can be used. All of the various types of antialiasing are discussed in detail in **"Blender Modes and Assumptions" on page 327**.

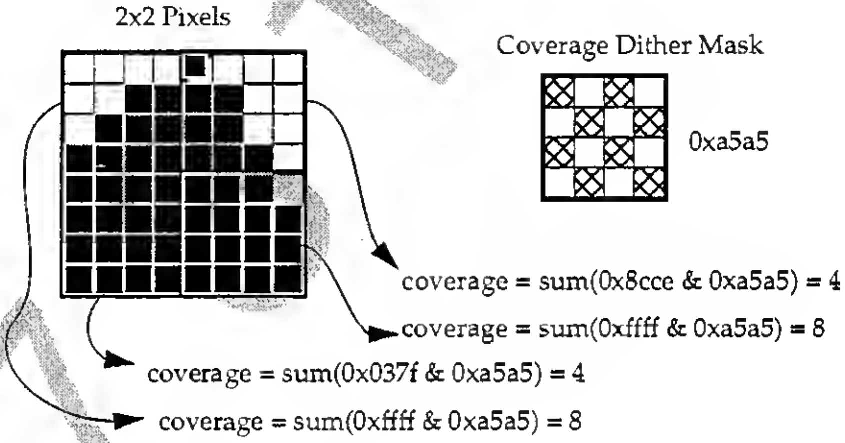
For each pixel, a subpixel mask is computed. This mask is a 4x4 grid of bits where the bit is one if the subpixel is covered by the primitive and zero if the subpixel is not covered. The mask is converted to a coverage value by adding all the bits of the mask together. Since we only have three bits of coverage, the sixteen subpixels must be dithered to eight. The coverage value is optionally combined with the pixel's alpha value. This is useful for antialiasing edges created by a texture cut-out. In the blender, the pixel color and the last value stored for the pixel in memory are combined. The blender also combines the pixel coverage and memory coverage and does z-buffering. The blender typically performs operations such as antialiasing the interior edges of objects and transparency. The new pixel's color, coverage, and Z are stored in the frame buffer. The Video Interface (VI) reads the pixel color and coverage and antialiases the silhouettes of objects.

We will now discuss each hardware unit in the antialiasing datapath in isolation, before considering how these units work together to render a complete image.

Coverage Unit

The coverage calculation, as described previously, produces a 4-bit number for each pixel that indicates how much of the pixel was covered by a primitive. For example, a value of 8 (1.0) indicates the pixel was fully covered. A value of 1 (0.125) indicates only one subpixel was covered. An example of the coverage calculation is shown in Figure 16-4, "Coverage Calculation," on page 306

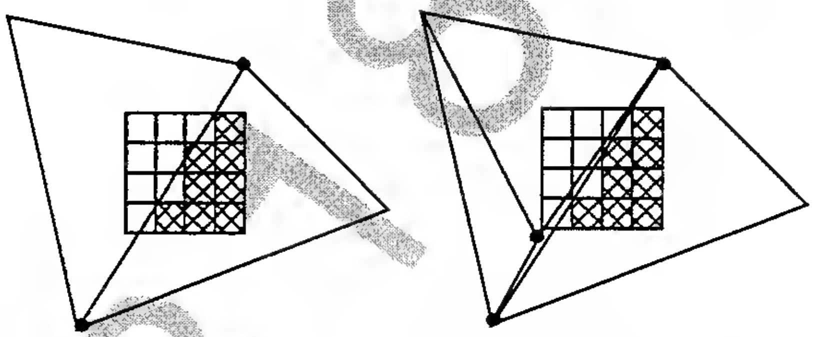
Figure 16-4 Coverage Calculation



2x2 Pixels
Coverage Dither Mask
0xa5a5
coverage = sum(0x8cce & 0xa5a5) = 4
coverage = sum(0xffff & 0xa5a5) = 8
coverage = sum(0x037f & 0xa5a5) = 4
coverage = sum(0xffff & 0xa5a5) = 8

Note that it is very important that primitives sharing an edge have complementary subpixel masks, otherwise cracks may appear between edges. In the RCP, if primitives use the same vertices to create the primitive, then the pixel mask will be complementary. There are, however, cases where bad modelling can lead to cracks, as in Figure 16-5, "Complementary Edges," on page 307. These cases can occur when (incorrectly) fractalizing terrain or (incorrectly) generating triangles from NURBs surfaces, for example.

Figure 16-5 Complementary Edges



Edges that share vertices will join correctly

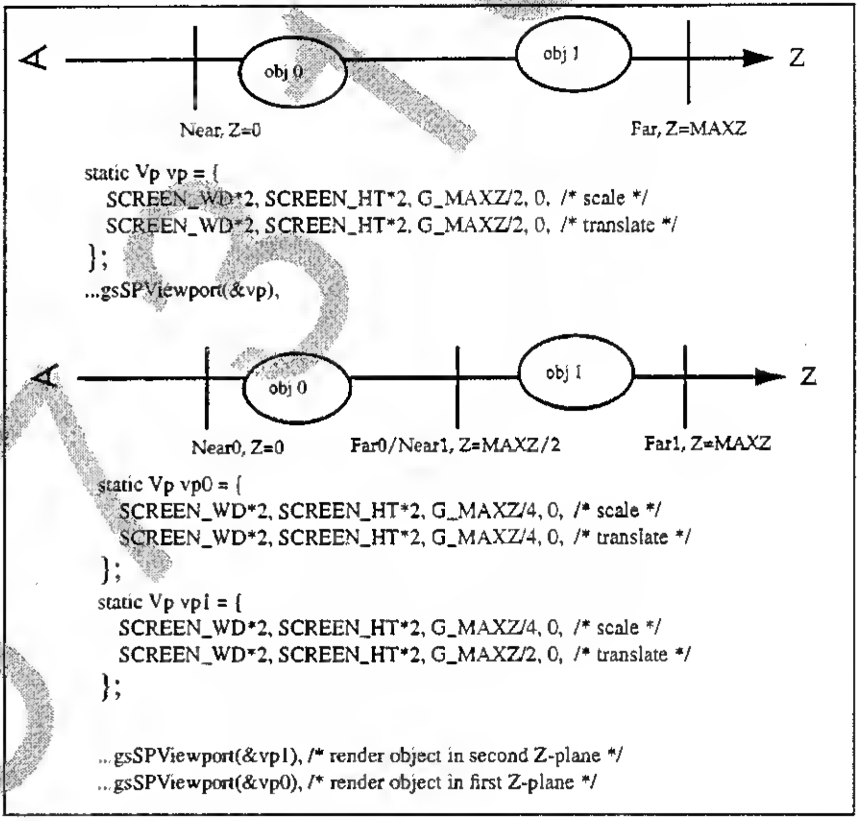
Edges that do not share vertices are not guaranteed to join correctly

1157389

Z Stepper

The Z-stepper calculates an 18-bit fixed point depth value (Z) for each pixel of a primitive. The value is of Z is normally zero at the near plane and maximum at the far plane, assuming a proper *g*SPViewport()* command. By manipulating the *g*SPViewport()* command, it is possible to split the z-buffer into separate *Z-planes,* see Figure 16-6, "Z-Buffer Planes," on page 308.

Figure 16-6 Z-Buffer Planes



```
static Vp vp = {
  SCREEN_WD*2, SCREEN_HT*2, G_MAXZ/2, 0, /* scale */
  SCREEN_WD*2, SCREEN_HT*2, G_MAXZ/2, 0, /* translate */
};
...gsSPViewport(&vp),
```

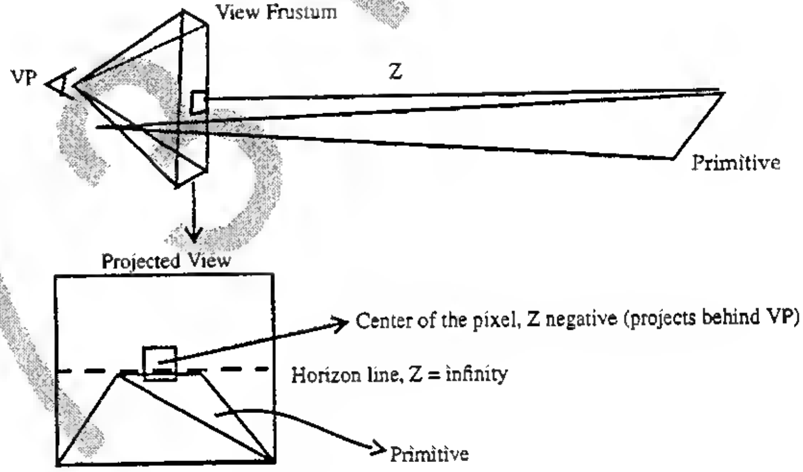
```
static Vp vp0 = {
  SCREEN_WD*2, SCREEN_HT*2, G_MAXZ/4, 0, /* scale */
  SCREEN_WD*2, SCREEN_HT*2, G_MAXZ/4, 0, /* translate */
};
static Vp vp1 = {
  SCREEN_WD*2, SCREEN_HT*2, G_MAXZ/4, 0, /* scale */
  SCREEN_WD*2, SCREEN_HT*2, G_MAXZ/2, 0, /* translate */
};

...gsSPViewport(&vp1), /* render object in second Z-plane */
...gsSPViewport(&vp0), /* render object in first Z-plane */
```

No attempt will be made to justify *why* one would do this, only that it is possible. Also, note that the *g*SPPerspNormalize()* command can be used to maximize Z precision. See Figure 12-2, "Perspective Normalization Calculation," on page 146 for more details about *g*SPPerspNormalize().*

There is also a source of constant Z (from a register) that can set using the *g*DPSetPrimDepth()* command. To select the constant depth, use the *g*DPSetDepthSource()* command. This may be useful when z-buffering sprites, for example.

The Z value is subpixel corrected so that it is always calculated on the primitive. To see why this is necessary, consider Figure 16-7, "Subpixel Correction of Z," on page 309:

Figure 16-7 Subpixel Correction of Z


View Frustum
VP
Z
Primitive
Projected View
Center of the pixel, Z negative (projects behind VP)
Horizon line, Z = infinity
Primitive

In this case, if you calculate Z at the center of the pixel, the Z value will be negative because Z will be projected behind the viewpoint. A better solution is to calculate the Z value at the subpixel, below the center of the pixel in this case, which intersects the primitive.

1157389

Blender

Color Blend Hardware

The blend mux selects input operands for the blender hardware. The controls for these muxes are in the RDP's *SetOtherModes* modeword. There are two sets of mux controls, one for each of the two possible rendering cycles.

The blend equation is of the form:

Equation 1 Blend Equation

$$color = \frac{(a \times p + b \times m)}{a + b}$$

The reasoning behind this equation will become evident in the discussion of the antialiasing algorithm discussed later in this document.

The four input operands (p, a, m, b) each have four possible sources so two bits are needed to control each mux. This gives a total of 8 bits per cycle of blend mux control. Since the pipeline can operate in one or two cycle mode (see *g*DPSetCycleType()*) the blender must select which of the sets of mux controls to use depending on the cycle type (G_CYC_1CYCLE or G_CYC_2CYCLE) and an internal cycle counter. The sources for the p and m muxes are identical and are shown in Table 16-1, "P and M Mux Inputs," on page 310.

Table 16-1 P and M Mux Inputs

Mux Select	Source
0	first cycle - pixel RGB, second cycle - blended RGB from first cycle
1	memory RGB
2	blend (register) RGB
3	fog (register) RGB

For select 0, the cycle select is built into the hardware. The 'blended RGB' refers to the **numerator** result of the blend equation, Equation 1, on the first cycle (it's fed back as an input). Note that this will only work if the **b** mux is set to **1.0 - a**, since only the numerator of the blend equation is provided to the input mux. **Register** RGBs refer to colors which can be set using the *g*DPSetFogColor()* and *g*DPSetBlendColor()* commands. Colors set using these commands are stored in registers within the RDP. Care must be taken to make sure that a *g*DPPipeSync()* command is issued previous to setting these registers. The *g*DPPipeSync()* command inserts a delay into the RDP pipe so that a previous primitive is guaranteed to be finished processing before the register is updated. It is anticipated that the user will set a group of attributes, process many primitives, set a new group of attributes, etc. The syncs are exposed to the user who can more likely determine the minimum number of syncs needed than would be possible in hardware. (Note that primitive color, *g*DPSetPrimColor()*, primitive depth, *g*DPSetPrimDepth()*, and scissor, *g*DPSetScissor()*, are attributes that do not require any syncs.

The sources for the a muxes are shown in Table 16-2, "A Mux Inputs," on page 311.

Table 16-2 A Mux Inputs

Mux Select	Source
0	color combiner output alpha
1	fog (register) alpha
2	(stepped) shade alpha
3	0.0

The sources for the **b** muxes are shown in Table 16-3, "B Mux Inputs," on page 311.

Table 16-3 B Mux Inputs

Mux Select	Source
0	1.0 - 'a mux' output
1	memory alpha

Table 16-3 B Mux Inputs

Mux Select	Source
2	1.0
3	0.0

In general, the RDP pipeline operates on RGBA pixels with 8 bits per component. The **1.0** in Table 16-3, "B Mux Inputs," on page 311 assumes the alpha is a number between 0.0-1.0. These numbers are actually fixed point and the output of the a and **b** alpha muxes have less resolution (5 bits) than the color components (8 bits) to reduce hardware cost. When this alpha is changing slowly across a face, Mach banding can occur due to the reduced number of discrete steps in the alpha channel.

Two dither commands can be used to reduce Mach banding effects: *g*DPSetColorDither(), and g*SetAlphaDither().* These commands basically add a small amount of randomness (1/2 of an LSB) to the color and/or alpha which makes the Mach banding less noticable. The *g*DPSetColorDither()* command also controls the dithering of RGB from 8 to 5 bits per component (for use in 5/5/5/1 pixel mode).

There are two variations of dithering that can be set using the *g*DPSetColorDither()* command. One is a *screen coordinate based dither* (G_CD_MAGICSQ or G_CD_BAYER) in which the dither matrix changes based on the location of the pixel on the screen. In other words, the dither pattern is registered to the screen. The *noise dither* (G_CD_NOISE), on the other hand, adds pseudo-random noise with a very long period into the LSBs of each pixel. In this mode, the dithering is not registered to the screen and will vary from frame to frame. Of course, you can disable color dithering altogether using the G_CD_DISABLE parameter.

Alpha dithering (*g*DPSetAlphaDither()*) for screen-based dither patterns uses the same matrix that is selected by the *g*DPSetColorDither()* command. However, the user may invert the pattern, G_AD_NOTPATTERN, or simply pass the pattern through unchanged, G_AD_PATTERN. The user may also select the noise pattern using G_AD_NOISE, or disable alpha dithering altogether using G_AD_DISABLE.

1153089

Note: The dithering of the RGB from 8 bits to 5 bits by adding 3 lsbs of noise to the original 8 bits (with clamping to prevent wrapping) is enabled even in 32 bit mode (8/8/8/8), where there is no truncation to be done. Since this one mode bit controls both RGB dither and alpha dither (which always is needed, even in 32 bit mode), opaque things should have the dither bit off in 32 bit mode (so the 3 lsbs don't get stepped on), but transparent things should have this bit on in 32 bit mode, since the noise from the alpha will be of the same order as the noise gratuitously added to the RGB.

Fog

Suppose we want to "fog out" from an image to a constant color as a function (set up in the RSP) of depth. We will assume the fog parameter is set up (per vertex) in the stepped alpha of the shaded triangle primitive (see "Vertex Fog State" on page 169). We will use the fog register color (*g*DPSetFogColor()*) as the color to fade too. We will use the stepped shade alpha as a control to determine how much of the fog color is used. The first cycle blend mux selects in Table 16-4, "Fog Mux Controls," on page 313 will achieve this effect.

Table 16-4 Fog Mux Controls

Mux	Source Selected
P	select 0, pixel RGB
A	select 2, stepped shade alpha
M	select 3, fog register color
B	select 0, 1.0 - stepped shade alpha

From the blend equation, Equation 1, you can see that these selects perform a linear interpolation between the fog color and the color combiner output color.

Equation 2 Fog Blend Equation

$$color = \frac{fogparam \times pixclr + (1.0 - fogparam) \times fogclr}{fogparam + 1.0 - fogparam}$$

The command *g*DPSetRenderMode()* is used to control these muxes as well as other blender modes. The command *g*DPSetRenderMode(*G_RM_FOG_SHADE_A, G_RM_FOG_SHADE_A*)* implements the mux controls for this fog effect in G_CYC_1CYCLE mode. Typically, this effect would be used only in G_CYC_2CYCLE mode, with the second cycle performing the blend of the pixel with memory. For example, *g*DPSetRenderMode(*G_RM_FOG_SHADE_A, G_RM_AA_ZB_OPA_SURF2*)* enables fog while rendering antialiased, z-buffered, opaque surfaces. In G_CYC_1CYCLE mode, only the fogging operation would be performed (no blend).

Coverage Calculation

From the previous discussion in "Coverage Unit" on page 306, coverage is a 4-bit value that indicates how many subpixels are occluded by a primitive. Note that a coverage of zero indicates that no subpixels were covered and the pixel does not need to be written to the frame buffer. Because there are only 3 bits of coverage available in the frame buffer, the coverage stored is actually:

Equation 3 Stored Coverage

$$memcvg = coverage - 1$$

When the pixel is read from memory, a one is automatically added to restore the actual coverage before it is used in calculations.

It is interesting to note that the Video Filter is concerned primarily with partially covered pixels around the silhouette edges of objects (see "Video

Filter" on page 326). Also, the antialiasing performed by the blender uses information about *coverage wraps*, i.e. when the sum of memory coverage and pixel coverage are greater than 1.0. Because of this, the frame buffer is initially cleared such that the coverage bits are all one, see "Color Image Format" on page 318.

Alpha Compare Calculation

From "Fill Mode" on page 180 and "Copy Mode" on page 180, you will notice that in G_CYC_COPY and G_CYC_FILL modes the blender hardware is bypassed and the fill color or image is written with no opportunity for read/modify operations.

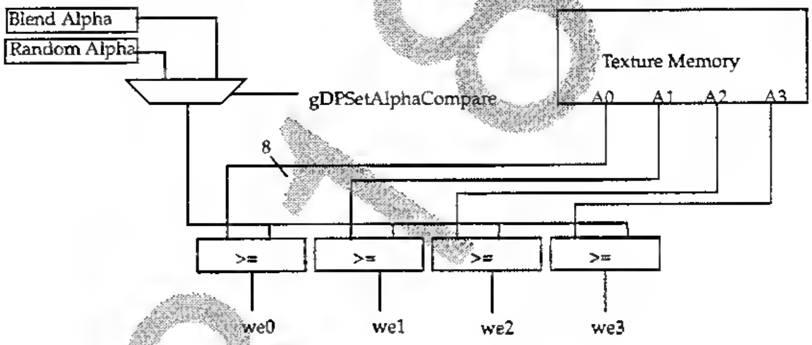
Note: When rendering in G_CYC_COPY or G_CYC_FILL, you should use the *RenderMode* G_RM_NOOP to make sure that reading of Z and color is disabled.

You can achieve a texture edge effect in G_CYC_COPY mode, however, by using the pixel alpha thresholded with the blend register alpha (*g*DPSetBlendColor()*). Figure 16-8, "Alpha Compare in Copy Mode for 8-bit Framebuffer," on page 316 shows that write enables are generated when the texel alpha is greater than or equal to blend alpha **for 8-bit framebuffers**. Also, note that for 16-bit RGBA texels there are no compares, the alpha bit simply acts as a write enable. Threshold alpha compare mode may be set by the following command: *g*DPSetAlphaCompare(G_AC_THRESHOLD)*.

1151389

Note: Alpha compare only works in G_CYC_COPY mode for the 16-bit RGBA color and 8-bit image types. You cannot copy the 32-bit RGBA color image type.

Figure 16-8 Alpha Compare in Copy Mode for 8-bit Framebuffer



Another alpha compare mode uses a hardware generated pseudo-random number as the threshold alpha. To set this mode, use *g*DPSetAlphaCompare(G_AC_DITHER)*.

Both G_AC_DITHER and G_AC_THRESHOLD can be used in G_CYC_1CYCLE or G_CYC_2CYCLE mode as well. In these modes, you can readily change the pixel's alpha from frame to frame, allowing various fade effects. In order to get the alpha of the pixel to the comparators, you must set the ALPHA_X_CVG and ALPHA_CVG_SEL bits properly. Figure 16-9, "Alpha Compare in One/Two-Cycle Mode," on page 317 shows a block diagram of the coverage/alpha combiner and alpha comparator logic. These controls are usually set as part of the *g*DPSetRenderMode* command. For example, the command *g*DPSetRenderMode*(G_RM_TEX_EDGE, G_RM_TEX_EDGE2) will do the right thing with these mode bits. See Table 16-6 for details on which bits are set for a particular *RenderMode*.

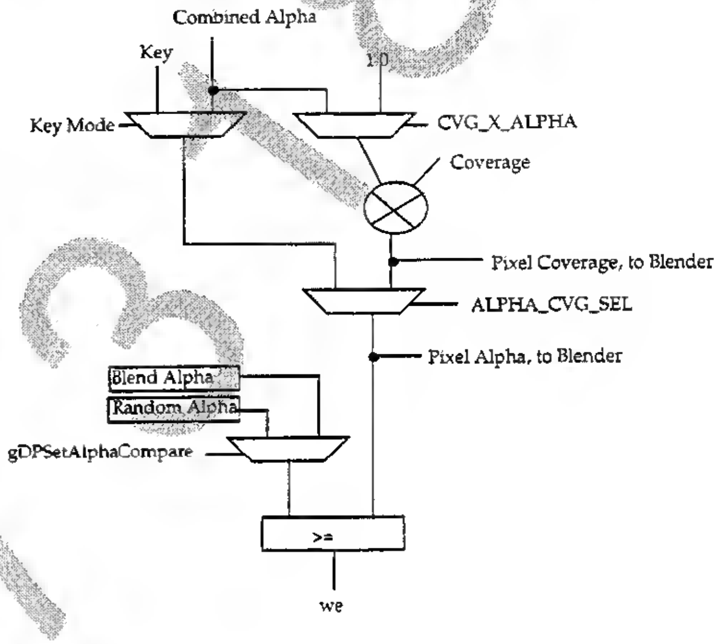
For rendering effects such as smoke, clouds, or explosions, set the texture alpha to the outline of the smoke orexplosion and render the texture onto a transparent polygon so that one can see through the smoke to the objects behind.

1153189

In this situation, the correct *g*DPSetRenderMode()* to use is G_RM_ZB_CLD_SURF or G_RM_CLD_SURF.

This 'cloud' mode preserves the antialiasing of objects behind the cloud primitive, unlike TEX_EDGE and XLU_SURF modes.

Figure 16-9 Alpha Compare in One/Two-Cycle Mode


Combined Alpha
Key
1.0
Key Mode
CVG_X_ALPHA
Coverage
Pixel Coverage, to Blender
ALPHA_CVG_SEL
Pixel Alpha, to Blender
Blend Alpha
Random Alpha
gDPSetAlphaCompare
>=
we

Blender ADD Mode

A special blender mode has been implemented that allows the pixel color to be added to the memory color:

```
#define RM_ADD(clk) \
    IM_RD | CVG_DST_SAVE | FORCE_BL | ZMODE_OPA |          \
    GBL_c##clk(G_BL_CLR_IN, G_BL_A_FOG, G_BL_CLR_MEM,     \
               G_BL_1)
#define G_RM_ADD        RM_ADD(1)
#define G_RM_ADD2       RM_ADD(2)
```

11573109

Several notes about this mode:

- You must set fog alpha equal to 0xff for this mode to work, e.g. gsDPSetFogColor(255, 255, 255, 255).
- Since the blender does not clamp the final color (all the inputs are clamped and normal interpolation operations won't under/over flow) the user must guarantee that the results will not overflow or "special effects" may occur.

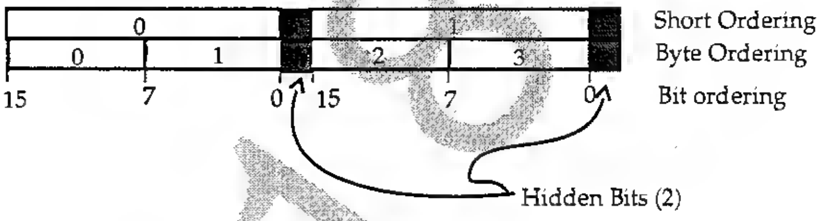
Color Image Format

The are three color image formats: 32-bit RGBA, 16-bit RGBA, and 8-bit. In addition, there are *hidden bits* that are available to the RDP memory interface but not readily visible to the programmer, see Figure 16-10, "Hidden Bits," on page 319. These hidden bits come from the fact that the RCP uses 9-bit RDRAMs. For 16-bit RGBA types, the hidden bits are used for storing coverage. For 32-bit RGBA types, the 3 coverage bits are stored as the 3 MSBs of the 8-bit alpha channel and the hidden bits are ignored. Note that the 32-bit RGBA mode does not provide increased alpha resolution. For 8-bit color images, the hidden bits are ignored.

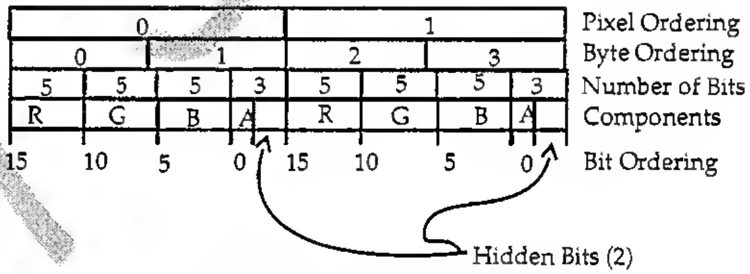
There hidden bits are logically the 2 LSBs of each 18-bit word. For memory accesses from other than the RDP memory interface (MI), only a 16-bit word is read/written. Other masters can indirectly set or clear the hidden bits by setting or clearing the LSB of the 16-bit word, respectively. For example, if the CPU writes the 16-bit binary value 10101010_10101010 to memory, the memory interface will actually write the 18-bit binary value 10101010_10101010_00. On the other hand, if the CPU writes the 16-bit binary value 01010101_01010101, the memory interface will actually write the 18-bit binary value 01010101_01010101_11.

115199

Figure 16-10 Hidden Bits


0
1
0
1
2
3
Short Ordering
Byte Ordering
15
7
0
15
7
0
Bit ordering
Hidden Bits (2)

Note: Hidden bits are only read/written directly by the RDP memory Interface. They are logically positioned as the LSBs of every 16-bit word, independent of Color Image type.


0
1
Pixel Ordering
0
1
2
3
Byte Ordering
5
5
5
3
5
5
5
3
Number of Bits
R
G
B
A
R
G
B
A
Components
15
10
5
0
15
10
5
0
Bit Ordering
Hidden Bits (2)

16-bit RGBA Format Showing Hidden Bits

Figure 16-11, "Color Image Formats," on page 320 describes the logical frame buffer formats.

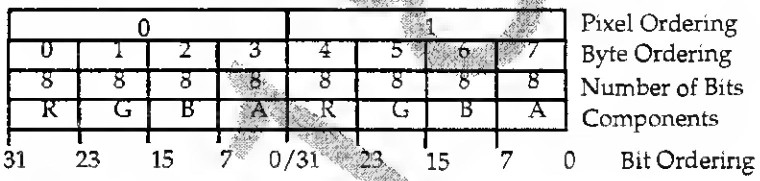
1157309

1157389

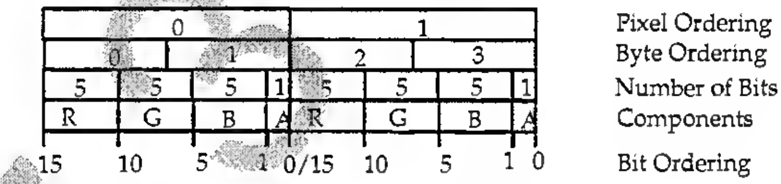
Image Alignment Requirements

The color image pointer, *g*DPSetColorImage()*, and the depth image pointer, *g*DPSetDepthImage()*, should be aligned to 64-bits, i.e. the 3 LSBs of the pointer should be zero.

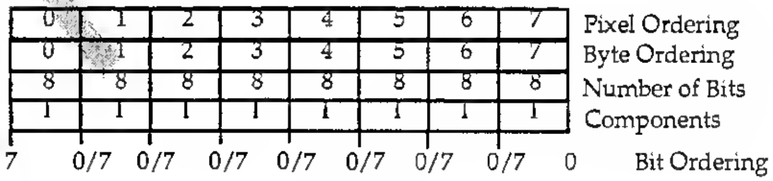
Figure 16-11 Color Image Formats


0
1
0 1 2 3 4 5 6 7
8 8 8 8 8 8 8 8
R G B A R G B A
31 23 15 7 0/31 23 15 7 0
Pixel Ordering
Byte Ordering
Number of Bits
Components
Bit Ordering

32-bit RGBA Format


0
1
0 1 2 3
5 5 5 1 5 5 5 1
R G B A R G B A
15 10 5 1 0/15 10 5 1 0
Pixel Ordering
Byte Ordering
Number of Bits
Components
Bit Ordering

16-bit RGBA Format


0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
8 8 8 8 8 8 8 8
I I I I I I I I
7 0/7 0/7 0/7 0/7 0/7 0/7 0/7 0
Pixel Ordering
Byte Ordering
Number of Bits
Components
Bit Ordering

8-bit I Format

Z Calculation

As mentioned in the "Z Stepper" section, *g*DPSetDepthSource()* selects the source of Z for the depth compares used in the z-buffer algorithm. This selects between primitive Z (a register), *g*DPSetPrimDepth()*, and stepped Z (from the triangle or line). *G*DPSetDepthSource()* also selects between primative DeltaZ (a register) and stepped DeltaZ. The 16 bit primitive Z register can supply the 15 integer bits of the Z value and the 16 bit deltaZ register can supply the 16 bits of the DeltaZ value.

For each z-buffered primitive, the change in Z per pixel change in the X and Y directions are calculated in the RSP as part of setup. These values are used in the z-buffer logic of the blender to create a composite **DeltaZ** for the pixel:

Equation 4 DeltaZ Calculation

$$DeltaZpix = |dZdx| + |dZdy|$$

DeltaZpix = | dZdx | + | dZdy |

The DeltaZ value is important in determining surface correlation-- that is, whether this pixel is part of the same surface as the pixel that is stored in memory. When computing whether the pixel is part of the same surface, the worst case **DeltaZ** is used:

Equation 5 Max DeltaZ Calculation

$$DeltaZmax = MAX(DeltaZpix, DeltaZmem)$$

The z-buffer compare equations are:

Equation 6 Max Z Test

$$MaxZ = (MemZ \equiv MAXZ)$$

Equation 7 Farther Compare

$$Farther = (PixZ + DeltaZmax) \geq MemZ$$

Equation 8 Nearer Compare

$$Nearer = (PixZ - DeltaZmax) \leq MemZ$$

Equation 9 In Front Compare

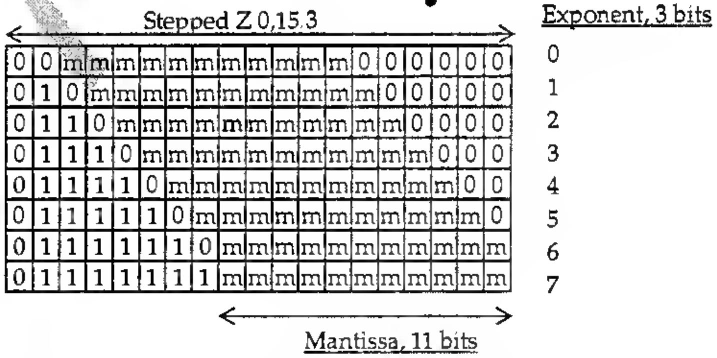
$$InFront = PixZ < MemZ$$

These signals are used along with coverage information to determine surface correlation for various antialiasing modes. See "Blender Modes and Assumptions" on page 327.

Z Image Format

The Z-buffer logic in the blender uses a fixed point, 0,15.3, 18 bit number for Z calculations. The delta Z is a 16 bit quantity that is used as a s15 number. The linear 18-bit Z that is stepped, is converted to a 14 bit floating point format before being stored. This encoding is shown in Figure 16-12, "Z Encoding," on page 322.

Figure 16-12 Z Encoding



																			Exponent, 3 bits
0	0	m	m	m	m	m	m	m	m	m	m	m	0	0	0	0	0	0	0
0	1	0	m	m	m	m	m	m	m	m	m	m	m	0	0	0	0	0	1
0	1	1	0	m	m	m	m	m	m	m	m	m	m	m	0	0	0	0	2
0	1	1	1	0	m	m	m	m	m	m	m	m	m	m	m	0	0	0	3
0	1	1	1	1	0	m	m	m	m	m	m	m	m	m	m	m	0	0	4
0	1	1	1	1	1	0	m	m	m	m	m	m	m	m	m	m	m	0	5
0	1	1	1	1	1	1	0	m	m	m	m	m	m	m	m	m	m	m	6
0	1	1	1	1	1	1	1	m	m	m	m	m	m	m	m	m	m	m	7

Three bits are stored for the exponent and 11 bits are stored for the mantissa. Here is some psuedo code for converting from the format stored in memory to the Z format used in calculations:

1151389

```
/*
 * Convert 11 bit mantissa and 3 bit exponent
 * to 0,15.3 number
 */
struct {
    int shift;
    long add;
} z_format[8] = {
    6, 0x00000,
    5, 0x20000,
    4, 0x30000,
    3, 0x38000,
    2, 0x3c000,
    1, 0x3e000,
    0, 0x3f000,
    0, 0x3f800,
};

    zvalue = (mantissa << z_format[exponent].shift) +
              z_format[exponent].add;
```

Notice that converting from a 18 bit fixed point number to a 14 bit floating point number, some precision may be lost. The lose of precision is greatest for small exponents. The highest precision is saved for large Z values, that is, for objects that are far away from the eye.

The DeltaZ is also encoded into 4 bit integer for storage into the Z-buffer using the following equation:

Equation 10 DeltaZ Encoding

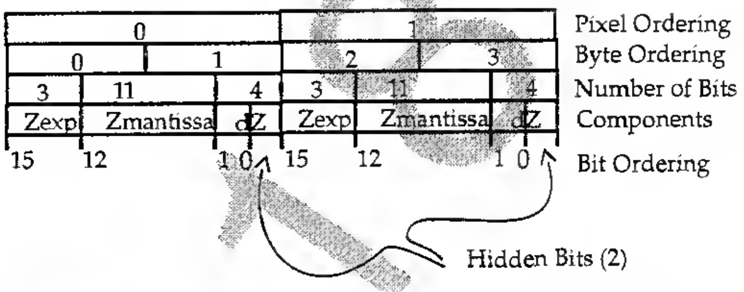
$$DeltaZmem = \log 2\,(DeltaZpix)$$

This is just a priority encoding of the DeltaZ value. The bit number of the most significant bit that has a value of one is stored.

11513189

The memory format for the Z and DeltaZmem is shown in Figure 16-13, "Z Memory Format," on page 324.

Figure 16-13 Z Memory Format

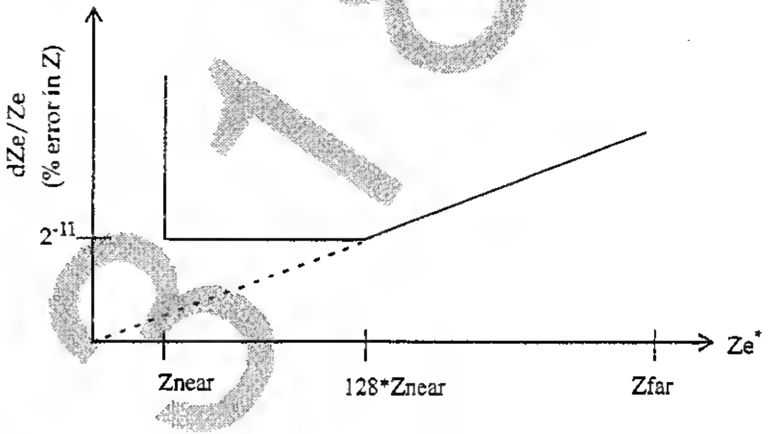

0
1
Pixel Ordering
0
1
2
3
Byte Ordering
3
11
4
3
11
4
Number of Bits
Zexp
Zmantissa
dZ
Zexp
Zmantissa
dZ
Components
15
12
1 0
15
12
1 0
Bit Ordering
Hidden Bits (2)

Note: Hidden bits are **only** read/written directly by the RDP Memory Interface. They are logically positioned as the LSBs of every 16-bit word.

Z Accuracy

The plot in shows the worst-case percent error in Z relative to the near and far planes.

Figure 16-14 Z Worst-Case Error



*Ze is eye-space Z.

11573189

Video Filter

The video filter performs the second pass of the analogizing algorithm. The first pass is done in the blender and involves antialiasing of *internal* or *non-silhouette* edges. After the image is rendered into the frame buffer, all pixels except those that are on the silhouettes of objects will be fully covered (coverage = 1.0). For *partially* covered pixels, the video filter performs a linear interpolation between the foreground color and the background color:

Equation 11 Video Filter Interpolation

$$OutputColor = cvg \times ForeGround + (1.0 - cvg) \times BackGround$$

The *ForeGround* color is always the color stored in the frame buffer for that pixel. The *BackGround* color is found by examining fully covered pixels in a 5x3 pixel area around the current pixel. Note that Z is not used in determining the *BackGround* color and so it is safe for Z to be single-buffered.

11573189

Blender Modes and Assumptions

Opaque Surface Antialiased Z-Buffer Algorithm, OPA_SURF

The main goal of this algorithm is to produce an antialiased rendering of polygonal surfaces without the need for sorting. The key to achieving this goal is to split the antialiasing problem up into several pieces, each of which is readily implemented.

There are basically three different kinds of antialiasing. The first is the antialiasing of textures within polygons. This is accomplished outside of the blender by the texture hardware, using the industry standard mipmapping technique. This uses tri-linear interpolation to produce a correctly sampled texture lookup. See "MIP Mapping" on page 232 for more details.

The second kind of antialiasing is the blending of polygon fragments within the pixels they share. The classic example of this is the pinwheel, where alternating black and white triangles meet at a center vertex. The pixel within which this vertex lies should be the average of the colors of all the triangles which share this vertex, weighted by the area of the pixel at the vertex covered by each of the triangles.

This blending is done in the blender hardware by computing Equation 1, where **p** is the color of the pixel of the new poly, **m** is the color of the pixel in the frame buffer memory, a is the coverage value of the new poly, and b is the sum of the coverage values of all the polygons already blended into that pixel in the frame buffer. Note that no matter what order the polygon fragments come in, they will all average in correctly.

The third kind of antialiasing is the blending of the silhouette of a foreground object against the background. This is traditionally done at rendering time in the blend unit. Unfortunately, doing it at this time has bad consequences for hidden surfacing.

Consider an internal edge of a surface (i.e., an edge shared by two visible polygons not at the silhouette). A priori, when the first of the two polygons is rendered, the blender does not yet know whether it is a silhouette edge (and hence needs to be blended with the background), or an internal edge

1153189

(and hence should not be blended with the background). Note that if an internal edge does blend with the background, there will be a line along the edge left when the second polygon blends with the first. Once the blending is done, there is no way to undo it. Also, note that the background may not even have been rendered yet, unless the rendering of polygons is done in depth-sorted order, which defeats the purpose of z-buffering.

The only way to deal with this is to postpone the blending of silhouette edges until after the whole scene is rendered. In fact, the final blending of the silhouette edges is done at display time by the video interface. While the details of this are beyond the scope of this document, the main point is that to do this blend on video output, there needs to be a coverage value left behind in the frame buffer, with which to interpolate between the foreground (the color of which is in the frame buffer) and the background (which is assumed to be in one or more of the neighboring pixels in the frame buffer). This interpolation is described in Equation 11.

Note that for this approach to work, we must be able to distinguish between internal edges within a surface and silhouette edges between an object and its background. This is only possible in the context of z-buffering. (If z-buffering is disabled, the internal edge blending must also be disabled, since we can no longer distinguish between internal and silhouette edges.)

In order to distinguish between an internal and a silhouette edge, we need in addition to the normal z-buffer containing depth information, some additional information so that we can tell if two polygons sharing a pixel are within the same surface or not. This added information is the slope of Z (depth) in screen space. This is computed as shown in Equation 4. The delta for the old polygon is stored in the frame buffer with the Z. The rule is then if the absolute difference in Z between the new polygon and the frame buffer is less than the max of the new DeltaZ and the frame buffer DeltaZ, then the new polygon is considered to be part of the same surface as the old polygon already in the frame buffer. If the new Z is clearly in front, it overwrites the frame buffer. If it is clearly behind, it is not written at all.

In fact, while this algorithm works as described above, it has some problems. First off, we are only representing one fragment per pixel. If there are multiple silhouettes within one pixel, there will be a slight artifact. There is some specialized hardware to reduce this effect (the divot circuit). However, some artifacts remain, and are simply tolerated.

The other, and considerably more visually obvious artifact is "punchthrough", where part of an object which should have been occluded "punches through" the object in front of it. This is caused by the z-buffer blending range being too large, usually due to large DeltaZ's from polygons that are very "edge on" to the viewpoint. There are two different mechanisms to prevent this artifact.

The first mechanism is to weight the weighting factors in the internal edge blend by how "edge on" they are. Polygons that are more "flat" are weighted more heavily than polygons that are more "edge on". Thus, the punching-through polygon is attenuated relative the polygon it is punching through.

The second mechanism to prevent punchthrough is to use the wrapping of the coverage value to distinguish between contiguous surfaces and a "new" polygon that is not part of that surface. Basically, if the coverage wraps (i.e., new cvg + old cvg > 1.0), then the new polygon must not be part of the previously rendered surface (or background). In that case, instead of using the DeltaZ range, the z-buffer does a strict compare between the new and old z, ignoring the deltas, since we know the new polygon is not part of the old surface.

Note: Note that the silhouette antialiasing part of this algorithm depends on not having shared edges across the silhouette (shared with the backfacing polygons adjacent to the silhouette). Consequently, back-facing polygons must be rejected (culled), or the coverage values at the silhouette edge will be incorrect for the display-time pass of the antialiasing algorithm. This is generally desirable in any case, since this saves the rendering time for the back-facing polygons, which should be invisible. Note that this is only a problem for closed polygonal surfaces (hulls), but not for "open" surfaces, like flags, which have "external" edges. So flag-like objects need to be represented in the display list twice, once frontfacing and once backfacing.

Transparent Surfaces, XLU_SURF

In addition to opaque surfaces, we would like to be able to do transparent surfaces with antialiasing and without the need to sort. There are two problems with this.

The first problem is avoiding sorting. Strictly speaking, this is impossible. In order for the colors to be correctly blended from multiple colored transparent surfaces, the surfaces need to be depth sorted (or carry around a lot of extra information, more than we have memory for), so we just don't do the right thing.

We do require all the transparent surfaces to be rendered after the opaque surfaces, but aside from that segregation, there is no sorting of the transparent (or opaque) surfaces. So multiple colored transparent surfaces will not be quite right. First off, this case doesn't come up much (most transparent surfaces are not colored, and it is rare for multiple transparent surfaces to line up). Secondly, even if it does, most people have had so little experience with multiple colored transparency that they don't know what to expect. Generally speaking, rendering the transparent surfaces in the same order, regardless of depth, looks just fine.

The second problem with transparency is internal edges. Here, we cannot do what we did in the opaque surface case. The pixels at an internal edge of a transparent surface are now blended with the (previously rendered, opaque) background, as are all the pixels in the interior of the transparent poly. So if we render one polygon sharing an internal edge, and then render the other polygon sharing that same edge, we must be sure not to blend any pixel twice, or there will be a noticable line on the internal edge as a consequence of blending twice. So we just don't blend internal edges of transparent surfaces.

In fact, this is a bit tricker than it seems. We still want the silhouette of a transparent object to be properly antialiased, so we need to be able to get the partial coverage values for the silhouette edges, without double blending the internal edges. This is done with a special mechanism provided just for transparency.

Under control of a special mode bit (CLR_ON_CVG), we can inhibit the writing of color (but not coverage) unless the coverage wraps (i.e., the sum of the old coverage in the frame buffer and the new coverage of the currently rendering polygon is greater than unity). On an internal edge of a transparent surface over a fully covered background, the first polygon will write the color, since full coverage plus any non-zero partial coverage must wrap. The coverage value is always written with the wrapped sum of the old pixel and new polygon coverage, which will be equal to the partial coverage of the new (first) poly. On the rendering of the second poly, however, the coverage values will sum to unity on the shared edge, which is not a wrap. So the second polygon will not write over the pixels on the shared edge of the first poly. Note that this works even if the underlying coverage is not unity (i.e., the transparent surface is over a pre-rendered silhouette edge), since still only one of the two transparent polygons sharing an internal edge will get to write (although it could be the second one instead of the first).

The blender in transparent surface mode uses a different form of the blend equation than for the opaque surface case. The blend equation for transparency is:

Equation 12

$$color = a \times p + (1.0 - a) \times m$$

where p is the color of the pixel of the new poly, m is the color of the pixel in the frame buffer memory, a is the opacity (alpha) of the new poly. Note that this can be obtained from Equation 1 by setting b=(1-a).

Note that since we never blend across an internal edge, we do not need to use the DeltaZ used to condition blending in the opaque surface case. Instead, we just compare Z directly, since the transparent surface can only be either clearly in front (in which case it is written with the transparency-blended color) or clearly behind (in which case it is not written at all, including coverage).

Note also that unlike opaque surfaces, which modify depth, transparent surfaces do not modify depth (although they do read it, to test for occlusion by a previously-rendered opaque object). This is because transparent surfaces do not want to prevent the writing of other transparent surfaces which are behind them (but in front of any opaque surfaces).

Transparent Lines, XLU_LINE

In this system, there is no explicit line generation hardware. So lines are rendered as degenerate polygons (i.e., a triangle two of whose sides are parallel, and whose third vertex is at infinity) using the normal triangle hardware. Rendering is very much like the rendering of surfaces. However, unlike surfaces, lines have no internal edges (since by definition, a line is an edge). So here, we don't have to worry about incorrectly blending internal edges at render time. So for lines, all the antialiasing is done at render time. Note, however, that as with transparent surfaces, lines must be rendered after any surfaces they may occlude. In fact, lines are considered intrinsically transparent. Opaque lines are simply transparent lines with an alpha of unity (or close to it).

The render-time antialiasing is done by multiplying the new polygon (line) coverage value with the alpha value, and using that as the alpha to do the transparency blending. This produces the correct result, due to the absence of internal edges.

The coverage value written into the frame buffer in line mode is the clamped sum of the old pixel coverage and the new line's coverage times its alpha. For nearly opaque pixels, the coverage will be clamped to unity, making any underlying silhouette edge not be modified by the video interface at the display-time part of the antialiasing algorithm. This prevents the overlying line from being disturbed by the underlying (and hence hidden) silhouette edge. However, if the coverage times alpha from the line is nearly zero, then the silhouette edge is not disturbed, since it should be visible through the line.

Lines do read depth, and thus can be occluded by opaque objects. However, lines, like transparent and decal surfaces, do not modify depth. They are thus blended in display list order, which for thin lines should not matter.

Note that "lines" need not be degenerate triangles. In particular, for a "ray" coming from somewhere in the foreground to a vanishing point at infinity, a normal triangle, with two vertices at the source of the ray, and the third at the vanishing point, produces the desired effect. Also note that these "rays" can be textured, to produce the effect of a diffuse particle beam (or "neon glow"), or even "tracer bullets" animated by changing texture coordinate mapping in the texture unit.

Texture Edge Mode, TEX_EDGE

Texture edge mode is the first of the special-purpose modes. It is a variation of opaque surface mode. It is intended mostly for 'billboard' type objects.

1153699

A textured 'billboard' uses alpha values of zero in the texture to define the outline of the tree. Either two billboards are crossed, or the one billboard moves to always face the eyepoint, so as to hide the two dimensional nature of the billboard. Frequently, only one bit of alpha (all or nothing) is available in the highly-packed texture modes usually used for billboards. Mipmapping can be used to maintain a properly antialiased tree texture, but at some point the eye can get close enough to the tree texture to exceed the highest level of detail. In this case the alpha will be interpolated over several pixels, creating a 'blurry' effect around the texture edges.

Texture edge mode simply allows the blurred alpha to be written as coverage. A blurryness in coverage does not produce a blurryness in the final image, since the backend filter simply ignores the internal partial coverage bits, recreating a sharp edge.

Decal Surfaces, OPA_DECAL, XLU_DECAL

In order to make the creation of models with complex details as simple as possible, we added a special mode to allow the rendering of 'decal' polygons (usually with a texture on them, like a flag or logo) over a previously rendered opaque surface. Unlike normal rendering, here we only want to render the decal if it is coplanar with the existing surface. Since we have the hardware to tell if a surface is (roughly) coplanar from the opaque surface blend case, we can use that to condition the writes of the decal. Otherwise the rendering is just like the opaque surface case. Here we rely on the opaque surface mechanism which conditions blends on the coverage value not wrapping. This insures that a decal polygon written over a fully covered surface will not blend with that surface, but will instead overwrite it. Internal edges of a decal will, however, be properly blended (with each other, but not with the underlying surface).

The coverage values of the decal surface wrap (as do opaque and transparent surfaces). Note that this only works well if the edge of the decal polygons do not coincide with a silhouette edge of the underlying surface. If this is the case, it would help to use clamping for coverage since this will result in simple aliasing. Using wrap in this case fails miserably, since the coverage values are double what they should be, with some of them wrapping and some of them not. However, even clamping is wrong. So decals should never be allowed to exactly coincide with a silhouette edge of the underlying surface.

Decal surfaces, like transparent surfaces do not modify depth, since they are supposed to be coplanar with the underlying surface, which already has the correct depth.

Note that there is also a transparent version of decals, for cases where some of the underlying surface should blend through. This uses the same decal z-buffering algorithm, but is otherwise like transparent surface mode.

Decal Lines, DEC_LINE

This mode also goes by the name "Tron mode", since its main effect is to exaggerate the polygonalness of an object, making it look more artificial, and hence more "hi-tech" (at least in the eyes of some artists). Like decal surfaces, the decal lines are only rendered if they are within the depth range of the underlying surface, which must be rendered before the decal line.

Aside from the different z-buffer algorithm, the only other difference between transparent lines and decal lines is the coverage written into frame buffer memory. For decal lines we do not modify coverage at all. This is so we do not disturb the antialiasing of the silhouette edges. Note that the half of the line which is "over the edge" of the silhouette will not be rendered. Consequently, while the inside edge of the decal line at the silhouette will be correctly antialiased at render time (as with transparent lines), the outside edge must still be antialiased at display time by the video interface. The coverage values at the silhouette are already correct before the decal lines are rendered. Internal edges are also already correct, since they are fully covered by the opaque surface rendering.

Note that the decal line case interacts poorly with one of the features of the video interface (the divot circuit). In particular, if a decal line is on the silhouette of an object, the divot circuit can disturb the decal lines at the silhouette. This can be avoided by not using decal lines anywhere they could be in the silhouette, or by turning off the divot circuit (at the loss of some antialiasing quality). Or it can simply be tolerated as it is. The effect is a thinning and breaking up of the decal line at the silhouette. In motion, the line doesn't scintillate much, and so is probably tolerable.

11537369

Interpenetration, OPA_INTER, XLU_INTER

Interpenetration is another special purpose mode, which allows antialiased interpenetration of polygons to a reasonable approximation, at the cost of some loss of protection against "punchthrough". This mode is intended for protrusions ("spikes") through a normal opaque surface, and for terrain, so the placement of objects (like trees) on the surface of the terrain need not be precise. Note that in the latter case, the terrain should be the interpenetrating surface, rendered last (after all the other opaque objects in the foreground). This ordering both prevents unnecessary punchthrough, as well as rendering more quickly (since the background terrain does not get written if it is behind an already rendered foreground object). Interpenetration mode should not be used for articulated joints, or other purposes where the interpenetration is used to connect what is supposed to be a contiguous surface. If it is used in this way, unacceptable punchthrough will result. It is probably better in these cases to use normal opaque surface mode if this is really necessary. The lines of intersection will alias, but if the two surfaces are roughly the same color, this may not be too noticable. Interpenetration mode should not be used gratuitously. There is both an opaque and transparent version of interpenetration mode.

The only down side of this is that interpenetration mode requires using the wrapping of coverage to select whether to do the coverage adjustment (if it wraps, and hence is a potentially interpenetrating surface) or not (if it doesn't wrap, and hence is assumed to be part of the same surface). This can result in unacceptable punchthrough if any previously rendered objects are behind and either very edge-on or very near the foreground interpenetration mode surface. This almost never happens for terrain (where an object is almost never both occluded and near the terrain surface), and is not terribly noticable in the case of small protrusions from a normal opaque surface object.

Note that interpenetrating polygons must be rendered after the surfaces which they interpenetrate (which need not themselves have been rendered in interpenetration mode). Other than that, there are no sorting requirements.

Particle System Mode, PCL_SURF

The so-called "particle system" mode is really just a clever use of the alpha dither compare function described above. This is not a true particle system, where a large number of discrete particles interact to produce some interesting effect (fire, explosions, water, etc.). This mode is just another polygonal rendering mode which can be used to make the surface of an object resemble the behavior of some kinds of particle systems. Note that this is much more efficient than a "true" particle system, since by this method, a large number of particles can be represented by a much smaller number of polygons. The remarkable thing about it is that it produces properly antialiased silhouettes with correctly rendered internal edges.

This mode is an odd hybrid of the normal 3D opaque surface mode and the 2D alpha dither compare mode. As described in "Alpha Compare Calculation" on page 315, alpha dither compare (G_AC_DITHER) is a way of getting "stipple transparency", on a pixel by pixel basis, by allowing a write of the pixel only if its alpha value is greater than the value of a random number between 0.0 and 1.0. This makes the probability of a write proportional to the alpha value, which averaging over many frames produces the effect of transparency. The most obvious use of this effect is a "transporter", where the object starts out opaque (alpha = 1.0), but then fades to nothing (alpha = 0.0) in a cloud of sparkles. With some other effects added in (textures, inverse transparency, etc.), this mode can also be used for explosions, fire, and the like. By animating the alphas with texture mapping, propagating "waves" of alpha can be produced. Due to the human visual system's predilection for finding patterns whether they are there or not (e.g., the "canals" on Mars), even though the "particles" are completely uncorrelated, the waves of alpha will create the perception of coordinated behavior among a large number of interacting particles.

In this mode, the interior of a polygon is strictly under the control of the alpha dither compare. The probability of a write is proportional to the alpha value. The silhouette edge is handled as for opaque surfaces, at display time in the video interface. The tricky thing is what to do about the internal edges of a surface.

Note that in this alpha dither compare case, the density of the neighborhood is a function of alpha. This means that on a shared internal edge, a blend will only be likely to occur if the alpha value is quite high. In fact, the probability of a blend is proportional to the square of the alpha value. If the blend doesn't happen, then the internal edge is treated like a silhouette edge, and as long as the neighborhood has enough uncovered pixels, the display-time antialiasing of these partially covered internal edge pixels will do the right thing. So the only possible problem is with internal edges at high alpha values, and here, the weighted average will just merge the (nearly identically colored) fragments from the two polygons with possibly the wrong weights. But since the two fragments are nearly identical, any error in weighting doesn't matter.

Blender Modes Truth Table

The *g*DPSetRenderMode()* macro sets all of the blender state necessary for different types of surfaces and antialiasing. The following tables map the *RenderMode* arguments to individual mode settings. The macro names used are from the *gbi.h* header file.

Mode Bit Descriptions:

AA_EN:	if not force blend, allow blend enable - use cvg bits
Z_CMP:	condition color write enable on depth comparison
Z_UPD:	enable writing of Z if color write enabled
IM_RD:	enable color/cvg read/modify/write memory access
CVG_DST[1:0]:	0) clamp if blend_en, new if !blend_en 1) wrap always 2) zap (force to full cvg) 3) save (don't overwrite memory cvg)
CLR_ON_CVG:	only update color on cvg overflow (transp surf)
CVG_X_ALPHA:	use alpha times cvg for pixel alpha and cvg
ALPHA_CVG_SEL:	use cvg (or alpha*cvg) for pixel alpha
FORCE_BL:	force blend enable
ZMODE:	0) opaque 1) interpenetrating 2) transparent 3) decal
alpha_compare_en:	condition color write enable on alpha compare, use the *g*DPSetAlphaCompare()* command to set.
dither_alpha_en:	compare alpha with pseudo-random noise (dithering), use the *g*DPSetAlphaCompare()* command to set.

Blender Mux Selects described in Table 16-1, "P and M Mux Inputs," on page 310, Table 16-2, "A Mux Inputs," on page 311, and Table 16-3, "B Mux Inputs," on page 311.

Note:

(1) Interpenetration is only meaningful in antialiased z-buffered mode.

(2) Always zap coverage in point sampled modes.

(3) If CLR_ON_CVG, must also FORCE_BL.

(4) If not CVG_X_ALPHA and ALPHA_CVG_SEL, must not FORCE_BL.

(5) Always FORCE_BL on non-z-buffered modes.

(6) In opaque surface mode, clamp/new CVG_DST mode works better on the edges of a decaled surface which closely corresponds to the edge of the underlying surface. Otherwise, use the wrap CVG_DST mode.

(7) To place new color regardless of other conditions, use FORCE_BL with p=don't care; m=pixel_color; a=zero; b=one; and don't enable Z_CMP.

Table 16-5 enumerates the recommended rendering modes for 3D graphics, discussed above in some detail. They are what the rendering engine was primarily designed to do. They produce the best visual quality at near-optimal efficiency.

Sub surface mode, SUB_SURF, is intended to be used as a way to get an opaque object upon which an antialiased transparent surface can be overlaid. The coverage values from the transparent surface will fill in the zapped coverage values from the initial opaque surface.

The terrain modes, *_TERR, are to get around the modification of the blending weights by DeltaZ, which was intended for punchthrough reduction. This causes aliasing of internal edges in cases where the object faces are non-coplanar. These new modes use the normal lerp blender mode, which is free of DeltaZ dependence, and hence doesn't alias. Note, however, that these modes do not handle "pinwheels" correctly, since they assume that only two polygons meet at any pixel, which is generally not true. But in the case of terrains, which have very large polygons, this is more nearly correct.

11573189

Table 16-5 Antialiased Z-buffered Rendering Modes, G_RM_AA_ZB

Mode	AA_EN	Z_CMP	Z_UPD	IM_RD	CVG_DST (0:clamp, 1:wrap, 2:zap, 3:save)	CLR_ON_CVG	CVG_X_ALPHA	ALPHA_CVG_SEL	FORCE_BL	ZMODE (0:opaque, 1:inter, 2:trans, 3:decal)	alpha_compare_en, g*DPSetAlphaCompare	dither_alpha_en, g*DPSetAlphaCompare	Blender Mux P	Blender Mux M	Blender Mux A	Blender Mux B
OPA_SURF	1	1	1	1	0	0	0	1	0	0	0	0	0	1	0	1
XLU_SURF	1	1	0	1	1	1	0	0	1	2	0	0	0	1	0	0
OPA_DECAL	1	1	0	1	1	0	0	1	0	3	0	0	0	1	0	1
XLU_DECAL	1	1	0	1	1	1	0	0	1	3	0	0	0	1	0	0
OPA_INTER	1	1	1	1	0	0	0	1	0	1	0	0	0	1	0	1
XLU_INTER	1	1	0	1	1	1	0	0	1	1	0	0	0	1	0	0
XLU_LINE	1	1	0	1	0	0	1	1	1	2	0	0	0	1	0	0
DEC_LINE	1	1	0	1	3	0	1	1	1	3	0	0	0	1	0	0
TEX_EDGE	1	1	1	1	0	0	1	1	0	0	0	0	0	1	0	1
TEX_INTER	1	1	1	1	0	0	1	1	0	1	0	0	0	1	0	1
SUB_SURF	1	1	1	1	2	0	0	1	0	0	0	0	0	1	0	1
PCL_SURF	1	1	1	1	0	0	0	0	0	0	1	1	0	1	0	0
OPA_TERR	1	1	1	1	0	0	0	1	0	0	0	0	0	1	0	0
TEX_TERR	1	1	1	1	0	0	1	1	0	0	0	0	0	1	0	0

11573839

Mode	AA_EN	Z_CMP	Z_UPD	IM_RD	CVG_DST (0:clamp, 1:wrap, 2:zap, 3:save)	CLR_ON_CVG	CVG_X_ALPHA	ALPHA_CVG_SEL	FORCE_BL	ZMODE (0:opaque, 1:inter, 2:trans, 3:decal)	alpha_compare_en, g*DPSetAlphaCompare	dither_alpha_en, g*DPSetAlphaCompare	Blender Mux P	Blender Mux M	Blender Mux A	Blender Mux B
SUB_TERR	1	1	1	1	2	0	0	1	0	0	0	0	0	1	0	0

Table 16-6 enumerates modes that are primarily for situations where the sorting by depth of a scene is trivial, for example, the terrain for a flight simulator (as long as it is not too mountainous). Otherwise, the cost of sorting the polygons by depth would be prohibitive. These modes can be mixed and matched with any of the other rendering modes, z-buffered or not. Note that for proper antialiasing, polygons should be rendered in forward painter's algorithm order (back to front), NOT inverse order. (This is NOT the "a-buffer" algorithm, which requires inverse painter's algorithm order.) So in a mixed rendering mode scene, any non-z-buffered background polygons should be rendered first.

Note that there is no decal surface mode. Since there is no Z to condition the blend, decal surface mode is identical to opaque surface mode. There is a decal line mode, since it is slightly different in the way it handles silhouette edges. Also since there is no z, there are no interpenetration modes.

The line modes are very similar to the z-buffered line modes, except that decal line mode zaps coverage to unity. This is because in the non-Z case, both sides of the line are rendered, and are already correctly antialiased at render time. For the non-line modes, blending is based on coverage wrap, since there is no Z to discriminate between new and contiguous surfaces.

Sub surface mode is intended to be used as a way to get an opaque object upon which an antialiased transparent surface can be overlaid. The coverage values from the transparent surface will fill in the zapped coverage values from the initial opaque surface.

The terrain modes are to get around the modification of the blending weights by DeltaZ, which was intended for punchthrough reduction. This causes aliasing of internal edges in cases where the object faces are non-coplanar. These new modes use the normal lerp blender mode, which is free of DeltaZ dependence, and hence doesn't alias. Note, however, that these modes do not handle "pinwheels" correctly, since they assume that only two polygons meet at any pixel, which is generally not true. But in the case of terrains, which have very large polygons, this is more nearly correct.

Table 16-6 Antialiased Non-Z-Buffered Rendering Modes, G_RM_AA

Mode	AA_EN	Z_CMP	Z_UPD	IM_RD	CVG_DST (0:clamp, 1:wrap, 2:zap, 3:save)	CLR_ON_CVG	CVG_X_ALPHA	ALPHA_CVG_SEL	FORCE_BL	ZMODE (0:opaque, 1:inter, 2:trans, 3:decal)	alpha_compare_en, g*DPSetAlphaCompare	dither_alpha_en, g*DPSetAlphaDither	Blender Mux P	Blender Mux M	Blender Mux A	Blender Mux B
OPA_SURF	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1
XLU_SURF	1	0	0	1	1	1	0	0	1	0	0	0	0	1	0	0
XLU_LINE	1	0	0	1	0	0	1	1	1	0	0	0	0	1	0	0
DEC_LINE	1	0	0	1	2	0	1	1	1	0	0	0	0	1	0	0
TEX_EDGE	1	0	0	1	0	0	1	1	0	0	0	0	0	1	0	1
SUB_SURF	1	0	0	1	2	0	0	1	0	0	0	0	0	1	0	1
PCL_SURF	1	0	0	1	0	0	0	0	0	0	1	1	0	1	0	0
OPA_TERR	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0

Mode	AA_EN	Z_CMP	Z_UPD	IM_RD	CVG_DST (0:clamp, 1:wrap, 2:zap, 3:save)	CLR_ON_CVG	CVG_X_ALPHA	ALPHA_CVG_SEL	FORCE_BL	ZMODE (0:opaque, 1:inter, 2:trans, 3:decal)	alpha_compare_en, g*DPSetAlphaCompare	dither_alpha_en, g*DPSetAlphaDither	Blender Mux P	Blender Mux M	Blender Mux A	Blender Mux B
TEX_TERR	1	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0
SUB_TERR	1	0	0	1	2	0	0	1	0	0	0	0	0	1	0	0

The point-sampled rendering modes in Table 16-7 are provided for completeness. They have no significant performance advantage over the antialiased modes. These modes can be mixed and matched with any of the other rendering modes, antialiased or not, and so could be used for "special effects" within an otherwise antialiased scene. Generally speaking, point sampling looks bad, and should be avoided.

Note that there is no distinction between point-sampled line and surface modes, since lines and surfaces only differ in the way they are antialiased. For the same reason there are no point-sampled interpenetration or texture edge modes.

For the point-sampled modes listed, coverage is usually zapped to unity to prevent the video interface from trying to antialias them. Note also that in these modes, because the coverage always wraps (since it is always fully covered to begin with), surfaces are never blended, and the DeltaZ range is never used in the z-buffering.

Cloud and overlay surface modes are versions of transparent surface and transparent decal surface which do not disturb coverage. These are intended as overlays, where the silhouette of the polygon will have zero opacity, and hence should not affect the antialiasing of the image. (Note that textures can still be bilerped, which is the only kind of antialiasing that matters in this case.

Table 16-7 Point-Sampled Z-Buffered Rendering Modes, G_RM_ZB

Mode	AA_EN	Z_CMP	Z_UPD	IM_RD	CVG_DST (0:clamp, 1:wrap, 2:zap, 3:save)	CLR_ON_CVG	CVG_X_ALPHA	ALPHA_CVG_SEL	FORCE_BL	ZMODE (0:opaque, 1:inter, 2:trans, 3:decal)	alpha_compare_en, g*DPSetAlphaCompare	dither_alpha_en, g*DPSetAlphaDither	Blender Mux P	Blender Mux M	Blender Mux A	Blender Mux B
OPA_SURF	0	1	1	0	2	0	0	1	0	0	0	0	0	1	0	1
XLU_SURF	0	1	0	1	2	0	0	0	1	2	0	0	0	1	0	0
OPA_DEC	0	1	0	0	2	0	0	1	0	3	0	0	0	1	0	1
XLU_DEC	0	1	0	1	2	0	0	0	1	3	0	0	0	1	0	0
CLD_SURF	0	1	0	1	3	0	0	0	1	2	0	0	0	1	0	0
OVL_SURF	0	1	0	1	3	0	0	0	1	3	0	0	0	1	0	0
PCL_SURF	0	1	1	0	2	0	0	0	0	0	1	1	0	0	3	2

The point-sampled, non-z-buffered rendering modes in Table 16-8 are provided for completeness. They have no significant performance advantage over the antialiased modes.

Since there is neither antialiasing nor z-buffering, there is no difference between lines and surfaces, and no such thing as interpenetration, decals, or

1157389

texture edges. Only the transparent surface mode requires the reading of the frame buffer at render time. The opaque modes simply overwrite the color and zap the coverage in the frame buffer.

Cloud surface mode, CLD_SURF, is a versions of transparent surface mode which does not disturb coverage. This is intended as an overlay, where the silhouette of the polygon will have zero opacity, and hence should not affect the antialiasing of the image. (Note that textures can still be bilerped, which is the only kind of antialiasing that matters in this case.

The ADD render mode adds the pixel color to the memory color. Note that you must set the fog alpha to 0xff for this mode to work, e.g. *gsDPSetFogColor(255, 255, 255, 255).* Since the blender does not clamp it's output values (all the inputs are clamped and the normal interpolation operations won't under/overflow) the user must guarantee that the results of the add operation will not overflow or weird results (effects?) may occur.

The NOOP mode is simply a mode that disables reading of color and Z and zeros the rest of the blender state. You should set this render mode when the cycle type is either G_CYC_FILL or G_CYC_COPY.

The PASS mode is used when the cycle type is G_CYC_2CYCLE. In this case you may not want to do anything on the first cycle but blend in the second cycle. An example is: *gsDPSetRenderMode(G_RM_PASS, G_RM_OPA_SURF).*

115139

1157389

Table 16-8 Point-Sampled Non-Z-Buffered Rendering Modes, G_RM

Mode	AA_EN	Z_CMP	Z_UPD	IM_RD	CVG_DST (0:clamp, 1:wrap, 2:zap, 3:save)	CLR_ON_CVG	CVG_X_ALPHA	ALPHA_CVG_SEL	FORCE_BL	ZMODE (0:opaque, 1:inter, 2:trans, 3:decal)	alpha_compare_en, g*DPSetAlphaCompare	dither_alpha_en, g*DPSetAlphaDither	Blender Mux P	Blender Mux M	Blender Mux A	Blender Mux B
OPA_SURF	0	0	0	0	2	0	0	0	1	0	0	0	0	0	3	2
XLU_SURF	0	0	0	1	2	0	0	0	1	0	0	0	0	1	0	0
TEX_EDGE	1	0	0	0	0	0	1	1	1	0	0	0	0	0	3	2
CLD_SURF	0	0	0	1	3	0	0	0	1	0	0	0	0	1	0	0
PCL_SURF	0	0	0	0	2	0	0	0	1	0	1	1	0	0	3	2
ADD	0	0	0	1	3	0	0	0	1	0	0	0	0	1	1	2
NOOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PASS	x	x	x	x	x	x	x	x	x	x	x	x	0	0	3	2

Creating New Blender Modes

There are two types of mode bits in the blender, cycle-dependent and cycle-independent. The blender mux controls are cycle-dependent since they may differ between cycle 0 and cycle 1. All the other mode bits in the blender do not change between cycle0 and cycle 1. The *g*DPSetRenderMode()* command is set up to take two arguments. See the discussion in “Antialiasing Modes” on page 204 for details on how to make calls with *g*DPSetRenderMode()*.

To define a new RenderMode you must create a new macro that takes the cycle number (1 or 2) as an argument. For example:

```
#define RM_AA_ZB_OPA_SURF(clk)                                  \
    AA_EN | Z_CMP | Z_UPD | IM_RD | CVG_DST_CLAMP |           \
    ZMODE_OPA | ALPHA_CVG_SEL |                               \
    GBL_c##clk(G_BL_CLR_IN, G_BL_A_IN, G_BL_CLR_MEM, G_BL_A_MEM)
```

This macro OR’s the mode bits that are not cycle-dependent together with the blender mux controls that are cycle-dependent. Next define two macros that instance the macro above for each clock cycle:

```
#define G_RM_AA_ZB_OPA_SURF      RM_AA_ZB_OPA_SURF(1)
#define G_RM_AA_ZB_OPA_SURF2     RM_AA_ZB_OPA_SURF(2)
```

To use this mode, you could make the following call:

```
gsDPSetRenderMode(G_RM_AA_ZB_OPA_SURF, G_RM_AA_ZB_OPA_SURF2)
```

Note: Creating new controls for the blender mux is fairly straightforward. Setting the other blender modes, however, presumes a detailed understanding of the hardware since many of these modes are interdependent.

Visualizing Coverage

As a special bonus render mode, we have added G_RM_VISCVG. This mode will display coverage in the frame buffer as gray-scale intensities. To use this mode:

1. Render you entire scene, but don’t send *FullSync* yet.
2. Send the following display list:

```
gsDPPipeSync(),
gsDPSetCycleType(G_CYC_1CYCLE),
gsDPSetBlendColor(255, 255, 255, 255),
gsDPSetPrimDepth(0xffff, 0xffff),
gsDPSetDepthSource(G_ZS_PRIM),
gsDPSetRenderMode(G_RM_VISCVG, G_RM_VISCVG2),
gsDPFillRectangle(0, 0, SCREEN_WD-1, SCREEN_HT-1),
```

Partial coverage will be displayed as darker shades of gray and full coverage will be displayed as almost white. Try experimenting with different antialiasing methods while visualizing the coverage to increase your understanding of these algorithms.

1157318 9

11573189

1157189

Chapter 17

Sprites

This chapter describes the use of Sprites. Sprites are rectangular images or textures that you draw on the screen. Large images must be drawn in small pieces called "tiles." Managing these pieces is the task of the Sprite Library and associated data structures. This chapter explains how to do simple things, such as clear the framebuffer with a specified image; and how to do complex things, such as draw multi-colored text or explosions.

Here is a simple outline for this chapter:

- Application Programmers Interface (API)
  - Making
  - Manipulating
  - Drawing
- Data Structures and Attributes
  - Bitmaps
  - Sprites
  - Attributes
- Tricks and Techniques
  - Sparse Sprites
  - Early Ending
  - Variable Size Bitmaps
  - Explosions
  - Bitmap Re-use
  - Sprite Re-use

- Examples
  Backgrounds
  Text (Fonts)
  Simple Game

11573189

Application Program Interface (API)

Making Sprites

Sprites are usually used to draw images onto the screen. For these simple cases, a few scripts are provided to automatically take a specified image and generate an appropriate sprite data structure. The generated sprite may then be edited manually or modified at run time to create dynamic behavior.

```
mksprite name imgfile.rgb tileX tileY overlap > sp_name.h
```

This program takes a Silicon Graphics image file and generates a sprite. This sprite consists of a number of individual bitmaps (tiles) that are tileX apart in the x direction and tileY apart in the y direction. If overlap is "0," then these bitmaps are exactly tileX by tileY in size and should not be scaled (see spScale()). If overlap is "1," then the tiles are (tileX+1) by (tileY+1) in size. These sprites may be scaled and the textures will be properly interpolated. This extra pixel of overlap, or "border," provides the required data to create smooth transitions between tiles. The generated file may be included in an application and the sprite may be manipulated with the name "name."

```
mkisprite name imgfile.rgb tileX tileY overlap > sp_name.h
```

This command is just like *mksprite*, except that it converts the image to an 8-bit Color Index format, computes the TLUT, and generates the sprite with all the appropriate changes to support this format.

Manipulating Sprites

void **spInit**(Gfx ***glistp*)

This routine is called at the beginning of sprite drawing. Some GBI display list commands are added to the specified glistp to get the RCP into the correct mode for sprite rendering. This sets default texturing modes.

void **spFinish**(Gfx ***glistp*)

This routine is called at the end of sprite drawing. Some GBI display list commands are added to the specified glistp to get the RCP to complete all pending drawing operations and reset the RCP to its regular state. It also tacks on a gEndDisplayList().

void **spMove** (Sprite **sp*, s32 *x*, s32 *y*)

This routine sets the screen position of the upper left-hand corner of the sprite.

void **spScale** (Sprite **sp*, f32 *sx*, f32 *sy*)

This routine sets the resizing amount for this sprite. Scales may be less than 1.0 to produce a smaller image, or greater than 1 to create an expanded image.

void **spSetZ** (Sprite **sp*, s32 *z*)

This routine sets the z-buffer depth of the sprite. This may cause the sprite to be obscured by previously drawn sprites that were drawn with a smaller value of Z.

void **spColor** (Sprite **sp*, u8 *red*, u8 *green*, u8 *blue*, u8 *alpha*)

This routine sets the color of the sprite. Based on how the sprite is to be drawn, this could be either the PRIMITIVE_COLOR or the FILL_COLOR.

void **spSetAttribute** (Sprite **sp*, s32 *attr*)

This routine sets the indicated attributes. "attr" can be the bit-wise OR of many attributes.

void **spClearAttribute** (Sprite **sp*, s32 *attr*)

This routine clears the indicated attributes. "attr" can be the bit-wise OR of many attributes.

void **spScissor** (s32 *xmin*, s32 *xmax*, s32 *ymin*, s32 *ymax*)

11573189

This routine specifies the bounding region in which sprites will be drawn. By default, this region is initialized with *xmin*=0, *xmax*=319, *ymin*=0, and *ymax*=239.

Drawing Sprites

Gfx ***spDraw** (Sprite **sp*)

This routine constructs a display list starting at sp->next_dl that draws the sprite into the framebuffer in the indicated way. This display list is terminated with an gEndDisplayList() entry, and the sp->next_dl entry is updated to point to one entry past this. The pointer to the start of this display list is returned.

11573899

Data Structures and Attributes

Bitmap Structure

Here is the actual structure of a single bitmap:

```
typedef struct bitmap {

    s16width;/* Size across to draw in texels */
    /* Done if width = 0               */

    s16width_img;/* Actual size across in texels */

    s16s;/* Horizontal offset into bitmap */
    /* if (s > width_img), then load only! */

    s16t;/* Vertical offset into base    */

    void*buf;/* Pointer to bitmap data        */
    /* Don't re-load if new buf     */
    /* is the same as the old one   */
    /* Skip if NULL */

    s16actualHeight;/* True Height of this bitmap piece */

    s16LUToffset;/* LUT base index (for 4-bit CI Texs) */

} Bitmap;
```

Sprite Structure

```
typedef struct sprite {

    s16x,y;/* Target position */

    s16width,/* Target size (before scaling */
    height;

    f32scalex,/* Texel to Pixel scale factor */
    scaley;

    s16expx, expy;/* Explosion spacing */

    u16attr;/* Attribute Flags */
    s16zdepth;/* Z Depth */
```

115731389

```
    u8red,/* Primitive Color */
    green,
    blue,
    alpha;

    u16startTLUT;/* Lookup Table Entry Starting index */

    s16nTLUT;/* Total number of LUT Entries */

    s16*LUT;/* Pointer to Lookup Table */

    s16istart;/* Starting bitmap index */

    s16istep;/* Bitmaps index step (see SP_INCY) */
    /* if 0, then variable width bitmaps */

    s16nbitmaps;/* Total number of bitmaps */

    s16ndisplist;/* Total number of display-list words */

    s16bmheight;/* Bitmap Texel height (Used) */

    s16bmHreal;/* Bitmap Texel height (Real) */

    u8bmfmt;/* Bitmap Format */

    u8bmsiz;/* Bitmap Texel Size */

    Bitmap*bitmap;/* Pointer to first bitmap */

    Gfx*rsp_dl;/* Pointer to RSP display list */

    Gfx*rsp_dl_next;/* Pointer to next RSP DL entry  */

} Sprite;
```

1157389

Attributes

Sprite attributes permit sprites to be used in a variety of different ways. The following detailed description of each attribute indicates how setting or clearing that attribute affects the appearance of the drawn sprite. Note also that these attributes are as independent as possible, thus greatly expanding the available variety and uses for sprites.

SP_TRANSPARENT

This attribute permits the Alpha blending of the sprite texture with the background.

SP_CUTOUT

Use alpha compare hardware to not draw pixels with an alpha less than the blend color alpha (automatically set to 1).

SP_HIDDEN

This attribute makes spDraw() on the sprite return without generating a display list.

SP_Z

This attribute specifies that z-bufferering should be on while drawing the sprite.

SP_SCALE

This attribute specifies that the sprite should be scaled in both X and Y by the amount indicated in scalex and scaley.

SP_FASTCOPY

This attribute indicates that the sprite should be drawn in COPY mode. This produces the fastest possible drawing speed for background clears.

SP_TEXSHIFT

This attribute indicates that textures are to be shifted exactly 1/2 texel in both s and t before drawing it. This creates a better antialiased edge along transparent texture boundaries when in cutout mode.

1157318 9

SP_FRACPOS

This attribute indicates that the *frac_s* and *frac_t* fields of the sprite structure are to be used to fine-position the texture into the drawn pixels.

SP_TEXSHUF

This attribute indicates that the tile textures have their odd lines pre-shuffled to work around a *LoadTextureBlock*(3P) problem. See the **Texture Mapping** chapter for more details on this problem.

SP_EXTERN

This attribute indicates that existing drawing modes are to be used rather than the sprite routines explicitly setting them.

11573189

Tricks and Techniques

Sparse Sprites

The buf in a bitmap entry may be NULL, indicating that nothing should be drawn. This area will be 100% transparent.

Early-Ending Sprites

Setting the width of a bitmap entry to zero (0) signals an early exit to drawing the sprite's bitmaps.

Variable Size Bitmaps

Each bitmap can have a different drawn "width" and the corresponding texture can have a different width_img. To vary the vertical size of a sprite, set the actual_height field. If this is bigger than the sprite's bmHeightReal, then this actual_height is used for loading TMEM.

Explosions

Each sprite can specify the spacing between tiles in pixels by setting the explx and exply fields. The default value is zero (0). This spacing is not affected by the scaling of the sprite.

Bitmap Re-use

If the buf of the current bitmap matches the buf of the previous bitmap (not counting NULL bufs) in this sprite, then TMEM will not be re-loaded. This very simple form of texture caching is used in the font example.

11573189

Sprite Re-use

Each sprite has an associated display list and an associated next_dl pointer. When spDraw is called, new display list entries are added to the area pointed at by next_dl. This doesn't have to correspond to the pre-allocated display list allocated for the sprite; it could point somewhere else.

This allows a sprite to get drawn multiple times, each with a different setting of some parameters (position, scale, color, solid/textured, and so on). Sufficient display list area must be allocated for this to operate correctly.

1157319

Examples

A sample sprite library demonstration program is provided in under /usr/src/PR/spgame. The demo shows how to use sprite library to do backgrounds, texts and a simple animation.

Backgrounds

Setting up copy mode. Using TLUTs to animate it.

Scrolling Background example (up/down, left/right)

Text (Fonts)

void text_sprite(Sprite *txt, char *str, Font *fnt, int xlen, int ylen)

This creates the appropriate bitmap to render the specified string in the indicated sprite. You can use a two-pass approach to render a larger number of characters.

Simple Game

Anyone for a quick game of pong? Explosions, animated textures. Too much fun!

1157389

Chapter 18

Sprite Microcode

1151789

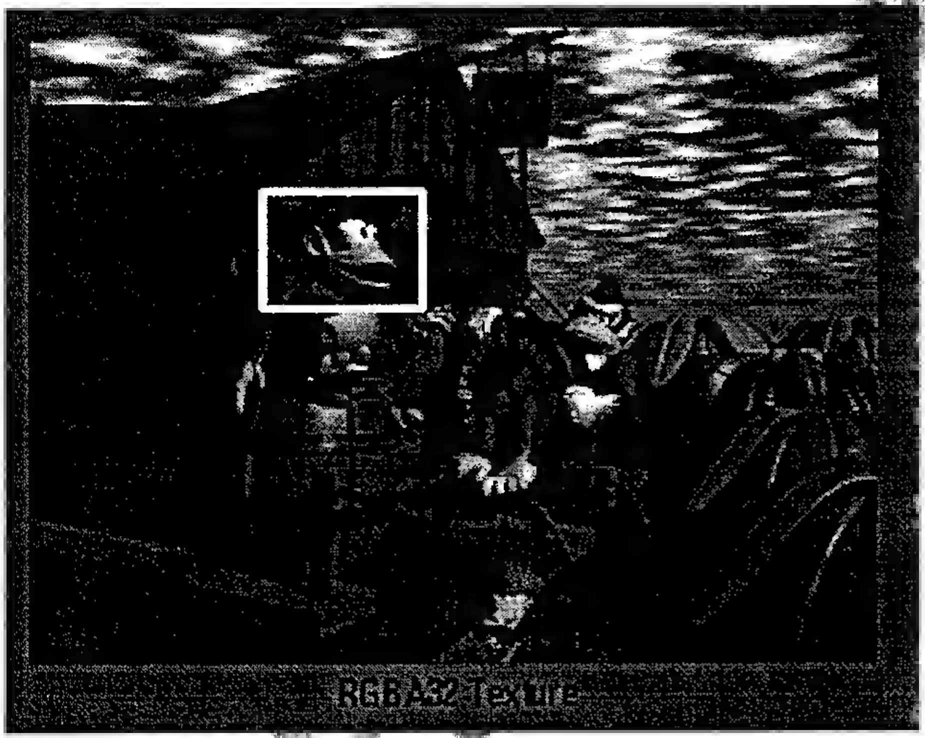
This chapter describes the use and operation of the Sprite Microcode, an alternative to the Sprite C Library described in the previous section.

The motivations for the creation of the Sprite Microcode were to provide an API which was more familiar to traditional 2D content developers, as well as offloading expensive calculations from the CPU to the otherwise largely idle RSP. By making use of the Sprite Microcode, applications gain access to additional CPU cycles per frame to perform game related computations.

The Sprite Microcode can co-exist with the Sprite Library in an application. Depending on the situation, either the Sprite C Library or the Sprite Microcode will be more appropriate at particular points in the game. One example where the Sprite C library would be more appropriate is for drawing text on the screen. An example where the Sprite Microcode would be more appropriate is the display of large textured background images which would require a large amount of CPU time by the Sprite Library to setup. The two APIs are also fairly different in their styles and the features they support. Developers are encouraged to try both methods to see which fits their needs more closely

Sprite Microcode Functionality

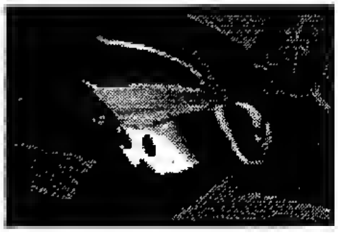
The functionality provided by the Sprite Microcode is the ability to display a subimage of arbitrary location and size out of a larger DRAM resident image of arbitrary texture type and size with optional scaling or mirroring in the X/Y axes.



Larger than 4K subimage

Large DRAM texture image

X/Y Scaled/mirrored screen image



Sprite Microcode API

The API provided for access to the Sprite Microcode is encapsulated into two new instructions illustrated by the following code fragment:

```
#include “gu.h”
#include “gbi.h”

uSprite MySprite;

guSprite2DInit(&MySprite, ImagePointer, TlutPointer,
               ImageWidth, RectangleWidth,
               RectangleHeight,
               ImageType, ImageSize,
               TextureScaleX, TextureScaleY,
               FlipTextureX, FlipTextureY,
               TextureStartS, TextureStartT,
               TranslateHorizontal, TranslateVertical);

gSPSprite2D(glistp++, OS_K0_TO_PHYSICAL(&MySprite));

Where MySprite is defined as a structure of type:


typedef struct {
  void *SourceImagePointer, void *TlutPointer,
  short Stride,
  short SubImageWidth, short SubImageHeight,
  char  SourceImageType, char  SourceImageBitSize,
  short ScaleX, short ScaleY,
  char  FlipTextureX, char  FlipTextureY,
  short SourceImageOffsetS, short SourceImageOffsetT,
  short PScreenX, short PScreenY,
  char dummy[2];
} uSprite_t;

typedef union {
  uSprite_t  s;
    long long int        force_structure_allignment[4];
} uSprite;
```

1157389

Where the parameters are defined as:

`SourceImagePointer` The address of the texture image in memory out of which a subrectangle is to be displayed

`TlutPointer` The address of an optional color index table for use with CI images. Use NULL for non-CI images

`Stride` The width in texels of the original base image in memory

`SubImageWidth` The width in texels of the subimage which is to be displayed

`SubImageHeight` The height in texels of the subimage which is to be displayed

`SourceImageType` The format of the texture image in memory. All supported hardware texture formats are allowed.

`SourceImageBitSize` The number of bits per texels of the input image. All supported hardware texture sizes are allowed.

`ScaleX`, `ScaleY` The s5.10 fixed point axis scaling ratios which are to be applied to the input image. A value of 1024 specifies 1 to 1 scaling. A value of 512 specifies that each input texel should be scaled up to 2 output screen pixels. Scale values should be <= 1024 in order to prevent sampling artifacts from occuring. Scale values must be positive. Use the `FlipTextureX` or `FlipTextureY` parameters to create negatively scaled images.

`FlipTextureX`, `FlipTextureY` Specifies whether the image should be mirrored in the X or Y direction before display

`SourceImageOffsetS`, `SourceImageOffsetT` The offset in texel rows or columns from the origin of the input base image where the texture subrectangle which is to be displayed starts

`PScreenX`, `PScreenY` Specifies the starting X or Y location in screen coordinates of the output image. The origin is in the upper left corner of the screen.

The `guSprite2DInit()` call merely copies its parameters into the passed in uSprite structure. The call can be eliminated if the application sets up the structure directly.

The Sprite Microcode automatically handles the division of the input subimage into 4K texture segments, loads them into TMEM and issues the appropriate RDP commands to setup and render a series of connected Texture Rectangles to display the subimage at the desired location and scaling. The Sprite Microcode keeps track of the s and t coordinates for the generated texture subRectangles.

The Sprite Microcode clamps the coordinates for the generated texture rectangles to prevent overflow of the RDP screen space registers. Texture Rectangles which have their X or Y starting values less than zero are clipped and their starting s and t texture coordinates adjusted so that they begin at the screen boundary. Texture rectangles which have their ending Y value less than zero or their starting Y value > 1023.75 are thrown away entirely.

More information about the Sprite Microcode can be found in the man pages for **gspSprite2D (3P)** and **guSprite2DInit (3P)**

1157389

11573189

PART

Ultra 64 Audio

11573189

11573189

Chapter 19

The Audio Library

11573189

The Nintendo 64 Audio Library is a lightweight library of functions. It provides game developers with the ability to interactively synthesize and manipulate audio on the Nintendo 64. It provides support for both sampled sound playback and Wavetable synthesis. This is accomplished with four software objects: the Sound Player, the Sequence Player, the Synthesis Driver, and the Audio Synthesis Microcode. These are shown in Figure 19-1, "Audio Software Architecture," on page 370.

- The Sound Player is useful for the playback of single sample sound effects or streamed audio. It is capable of playing back either ADPCM compressed sounds, or uncompressed 16 bit sound.
- The Sequence Player can exist in either of two types. The first type plays back Type 0 MIDI sequence files and the second type plays back a format of compressed MIDI unique to the Nintendo64. In both cases, the sequence player handles sequence, instrument bank, and synthesizer resource allocation, sequence interpretation, and MIDI message scheduling.

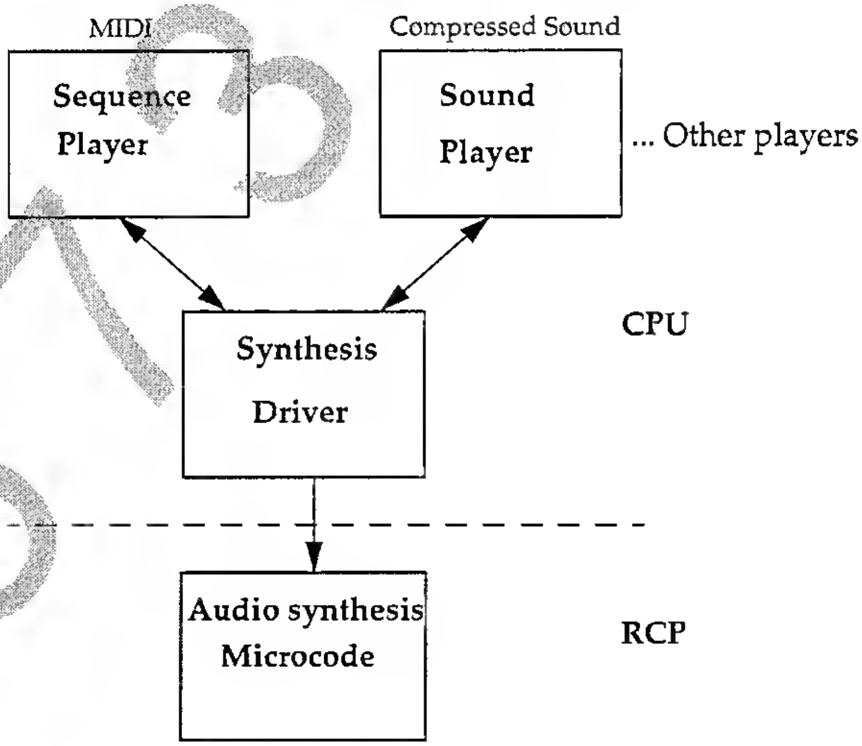
Note: Both the Sequence Player and the Sound Player are clients of the Synthesis Driver. The Driver can support an arbitrary number of clients, including multiple Sound and Sequence Players.

- The Synthesis Driver is responsible for creating audio Command Lists, which are packaged into tasks by the Application program and passed on to the Audio Synthesis Microcode. It allows Driver clients to assign wave tables to synthesizer voices, and control the playback parameters.

- The Audio Synthesis Microcode processes the tasks passed to it by the application and synthesizes stereo 16- bit samples, which the application in turn passes to the Audio DACs.

This chapter contains descriptions of the Sound Player, Sequence Player, and Synthesis Driver APIs . Many application programmers will be satisfied with the interfaces provided by the Sound and Sequence Players. Most of the Synthesis Driver API is intended for programmers who want to create their own players (see the section titled "Writing Your Own Player" for more information); however, all programmers should understand certain functions essential for the creation of audio Command Lists.

Figure 19-1 Audio Software Architecture


MIDI
Compressed Sound
Sequence Player
Sound Player
... Other players
Synthesis Driver
CPU
Audio synthesis Microcode
RCP

1157159

The following sections outline the data structures and API calls that are necessary to make use of the audio library. Further details on some of the data structures can be found in Chapter 15. The data structure definitions and function prototypes for the calls described are in the include file *libaudio.h,* which is part of the software release. Also included as a part of the software release are reference (man) pages for each of the function calls.

1157318

Generating Audio Output

The basic process for generating, and playing audio can be summed up by the following steps.

1. Create and initialize the neccessary resources. (Typically, an audio heap, a synthesizer, and a player)
2. Repeatedly make calls to alAudioFrame to generate the audio task lists.
3. Execute these audio tasks lists on the RSP.
4. Set the output DAC's to point to the audio output, with a call to osAiSetNextBuffer().

The creation and initialization of the neccessary resources is somewhat dependent on your applications needs, but typically you will need to take the following steps.

1. Create an audio heap with a call to alHeapInit.
2. Set the hardware output frequency with a call to osAiSetFrequency.
3. Create a synthesizer with a call to alInit(). (alInit will require that you have a callback routine to initialize the audio dma structures)
4. Create message queues for receiving signals that allow you to time your audio processing.
5. Create a player, (such as a sound player or sequence player) to sign into the synthesizer.
6. Initialize the resources specific to the player(s) that you have created.

11573189

Sampled Sound Playback

Representing Sound

The Audio Library supports playback of both uncompressed and ADPCM compressed, 16-bit audio. An audio waveform is represented with the Sound object via the ALSound structure. The `ALSound` structure contains entries for the Envelope, Pan, and Volume, along with a pointer to the `ALWaveTable` structure (which contains the audio).

Collections of sounds can be stored in an `ALBankFile` structure. The format of this structure is described in Chapter 21, "Audio File Formats". The tools available to create Bank Files for inclusion in the ROM are described in Chapter 20, "Audio Tools".

Note: Currently, the only supported sample formats are single-channel, ADPCM compressed and 16-bit uncompressed.

Playing Sounds

The Sound Player is the mechanism by which the Audio Library plays back individual sounds, such as isolated sound effects. It is responsible for allocating the resources needed to play a sound and for controlling the performance of the sound data for the application.

There are certain steps you must take for your game to play a sound. At a minimum, you must:

1. Create and initialize the basic resources described in the section Generating Audio Output.
2. Instantiate the Sound Player with alSndpNew(). The Sound Player created also signs in as a client to the Synthesis Driver.
3. Copy the sound bank's .ctl file into RAM, and initialize it with a call to alBnkfNew.
4. Allocate a sound with a call to alSndpAllocate().
5. Set the Sound Player's target sound to reference your sound with alSndpSetSound().

1157399

6. Play the sound with alSndpPlay().
7. Stop the sound when you are finished with alSndpStop(). Note that if the sound is not looped, the sound player will take care of stopping the sound when it is finished playing. However, you can stop the sound at any time during playback with this call.

When the sound is no longer needed, the resources in the Sound Player can be freed with a call to alSndpDeallocate(). If the Sound Player itself is no longer required, it can be removed from the Synthesis Driver client list with alSndpDelete().

The Sound Player can play both looped and unlooped sounds. When playing a sound, the Sound Player steps through the Envelope states Attack, Decay, and Release. Envelope parameters are defined in the `ALSound` structure. The duration of the sound is determined by the sum of the Attack time, Decay time, and Release time, or the length of the wave table (whichever is shorter), scaled by the pitch.

For looped sounds, the duration is always determined by the Envelope parameters and the pitch. If the Envelope Decay time is set to -1, the sound will continue playing (that is, it will never enter the Release phase) until it is stopped by the application with a call to alSndpStop(). Envelope times are scaled by the playback pitch so that regardless of pitch, finite-length sounds play to completion. For example, by default, a sound played an octave lower plays for twice as long as it does at unity pitch. Loop points for sounds are embedded in the `ALWaveTable` structure. (Loop points will be automatically extracted from the .aiff file when using the file conversion tools provided.)

Various parameters that affect the playback of a sound can be set before and during playback. When a sound is allocated to a Sound Player, an ID is returned that uniquely identifies that sound. Parameters for a particular sound are set by first setting the target sound with a call to alSndpSetSound(), and then making a subsequent call to set a parameter for the target sound. Available calls are detailed in Table 13-1.

Note: Each sound allocated to a Sound Player has a unique ID and private parameter values and play state. To play the same sound simultaneously, possibly with different parameter settings, it must be allocated multiple times to the Sound Player.

A summary of Sound Player functions is given below. Details can be found in the reference (man) pages.

Table 19-1 Sound Player Functions

Function	Description
alSndpNew	Creates a new Sound Player.
alSndpDelete	Removes a Sound Player from the Synthesis Driver's client list.
alSndpAllocate	Allocate a sound to a sound player.
alSndpDeallocate	Deallocate a sound from the sound player.
alSndpSetSound	Sets the Sound Player's current sound.
alSndpGetSound	Returns the Sound Player's current sound.
alSndpPlay	Plays the Sound Player's current sound.
alSndpPlayAt	Plays a sound at some specified time in the future.
alSndpStop	Stops the current sound from playing.
alSndpGetStates	Gets the current state (stopped or playing) of the current sound.
alSndpSetPitch	Sets the pitch for the current sound.
alSndpSetVol	Sets the playback volume of the current sound.
alSndpSetPan	Sets the pan position of the current sound.
alSndpSetPriority	Sets the sounds priority value.
alSndpSetFXMix	Sets the wet/dry mix of the current sound.

115A3199

Sequenced Sound Playback

You will be concerned with three issues when using sequenced sound on the Nintendo 64:

- representing the sequence data
- representing the instruments or sounds that make up the sequence
- controlling the sequence playback

Representing the Sequence

The Audio Library supports two different sequence players. The first sequence player uses Type 0 MIDI sequences. Sequences are represented at runtime with the `ALSeq` structure. This structure encapsulates sequence data that conforms to the Standard MIDI Files 1.0 specification for Type 0 MIDI files. The Type 0 MIDI file format contains a time-ordered MIDI message that specifies music events. It is described in detail in the "Standard MIDI Files 1.0" specification published by the MIDI manufacturers association.

The second sequence player uses a compressed format of sequence data unique to the Nintendo64. This format is detailed in Audio Formats chapter. Sequences are represented at runtime with the ALCSeq structure. Besides differences in the format of the data, the compressed MIDI sequence player handles loops in a different fashion and does not support markers.

To use a Type 0 MIDI sequence in your game, you must first initialize an `ALSeq` structure with alSeqNew(). To use the compressed MIDI sequence player, you first initialize an ALCSeq structure with alCSeqNew(). After initializing the `ALSeq` structure, you can perform sequence operations.

The alSeqNextEvent() call returns the MIDI event at a specified location in the sequence. The alSeqNewMarker() call creates a sequence position marker that can be used in conjunction with the Type 0 Sequence Player to set playback time and loop points. The convenience functions alSeqTicksToSec() and alSeqSecToTicks() convert between seconds and MIDI clock ticks.

Note: Normally, you won't call alSeqNextEvent() directly, because it is called by the Sequence Player during sequence playback.

The sequence calls are described in detail in the reference (man) pages. Brief descriptions are given in Table 13-2.

Table 19-2 Sequence Functions

Type 0 MIDI Sequence Player Function	Compressed MIDI Sequence Player Function	Description
alSeqNew	alCSeqNew	Initializes the sequence control structure.
alSeqNextEvent	alCSeqNextEvent	Returns the next MIDI event from the sequence.
alSeqNewMarker	alCSeqNewMarker	Initializes a marker for a given event time.
alSeqGetLoc	alCSeqGetLoc	Sets a marker to the sequence's current location.
alSeqSetLoc	alCSeqSetLoc	Sets the sequence to the location specified by the marker.
alSeqTicksToSec	alCSeqTicksToSec	Converts a time value from MIDI clock ticks to microseconds.
alSeqSecToTicks	alCSeqSecToTicks	Converts a time value from microseconds to MIDI clock ticks.

Representing Instruments

Instruments are represented at runtime by the `ALBankFile` structure. This structure describes the instruments that sound in response to an event in the sequence. Bank Files are composed of Banks; which are composed of Instruments; which themselves are composed of groups of Sounds, KeyMaps, Envelopes, and gain and pan information. The Bank File format is described in detail in the Audio Formats chapter.

To use a Bank File in your game, you must first create a runtime structure to represent it. This is accomplished with the alBnkfNew() function (See Table 13-3). Both sequence players use the same function call for this operation.

Table 19-3 Bank Functions

Type 0 MIDI Sequence Player Function	Compressed MIDI Sequence Player Function	Description
alBnkfNew	alBnkfNew	Initializes a collection of banks for use with a Sequence Player.

Playing Sequences

The Sequence Player is the mechanism by which the Nintendo 64 Audio Library plays back MIDI sequence files. It is responsible for allocating the hardware and software resources needed to play a sequence and for controlling the performance of the sequence data for the application.

Note: A Sequence Player can play only one sequence at a time.

There are certain steps you must take for your game to play a music sequence. The minimum steps needed to use the Type 0 MIDI sequence player are listed below. Using the compressed MIDI sequence player is identical, only you use the calls specific to the compressed MIDI sequence player.

1. Create and initialize the basic resources described in the section Generating Audio Output.
2. Initialize the sequence by using alSeqNew().
3. Copy the bank file's .ctl file into RAM, and initialize the bank by using alBnkfNew().
4. Initialize the sequence player by using alSeqpNew().
5. Set the sequence player's bank by using alSeqpSetBank().
6. Set the sequence player's target sequence by using alSeqpSetSeq().
7. Play the sequence by using alSeqpPlay().
8. Stop the sequence when you are finished with it, by using alSeqpStop().

9. If the sequence player is no longer needed it can be removed from the Synthesis Driver's client list by using alSeqpDelete().

Table 19-4 Sequence Player Functions

Type 0 MIDI Sequence Player Function	**Compressed MIDI Sequence Player Function**	**Description**
alSeqpNew	alCSPNew	Initializes a Sequence Player.
alSeqpDelete	alCSPDelete	Removes a Sequence Player from the Synthesis Driver's client list.
alSeqpGetState	alCSPGetState	Returns the current state of the Sequence Player.
alSeqpSetBank	alCSPSetBank	Assigns a bank of instruments to the sequence.
alSeqpGetSequence	alCSPGetSequence	Gets a reference to the sequence that is currently bound to the Sequence Player.
alSeqpSetSequence	alCSPSetSequence	Makes the specified sequence the target sequence.
alSeqpPlay	alCSPPlay	Starts the target sequence playing.
alSeqpStop	alCSPStop	Stops the target sequence if it is playing.
alSeqpGetTempo	alCSPGetTempo	Returns the current playback tempo for the target sequence.
alSeqpSetTempo	alCSPSetTempo	Sets the current playback tempo of the target sequence.
alSeqpGetVol	alCSPGetVol	Returns the overall volume for the sequence.
alSeqpSetVol	alCSPSetVol	Sets the overall volume for the sequence.
alSeqpGetChlPan	alCSPGetChlPan	Gets the pan on the specified MIDI channel.

1157389

Table 19-4 Sequence Player Functions

Type 0 MIDI Sequence Player Function	Compressed MIDI Sequence Player Function	Description
alSeqpSetChlPan	alCSPSetChlPan	Sets the pan for the specified MIDI channel.
alSeqpGetChlVol	alCSPGetChlVol	Gets the volume for the specified MIDI channel.
alSeqpSetChlVol	alCSPSetChlVol	Sets the volume for the specified MIDI channel.
alSeqpGetChlProgram	alCSPGetChlProgram	Returns the program assigned to the specified MIDI channel.
alSeqpSetChlProgram	alCSPSetChlProgram	Assigns the given program to the specified MIDI channel.
alSeqpGetChlFXMix	alCSPGetChlFXMix	Gets the wet/dry FX mix on the specified MIDI channel.
alSeqpSetChlFXMix	alCSPSetChlFXMix	Sets the wet/dry FX mix on the specified MIDI channel.
alSeqpGetChlPriority	alCSPGetChlPriority	Gets the priority value for the specified MIDI channel.
alSeqpSetChlPriority	alCSPSetChlPriority	Sets the priority value for the specified MIDI channel.
alSeqpLoop	(Not Supported)	Sets the loop points for the target sequence.
alSeqpSendMidi	alCSPSendMidi	Sends the specified MIDI message to the sequence player.

Loops in Sequence Players

The way in which loops are handled in the sequence players is different. When using the Type 0 MIDI sequence player, the programmer must create a marker at the loop start point, and a marker at the loop end point. Then the sequence can be looped between these two markers using alSeqpLoop(). Using the compressed MIDI sequence player, loops are constructed by the

1157389

musician, in the tracks of the sequence by inserting controllers. (This is discussed in the chapter "Using the Audio Tools"). This method allows different loops for different tracks, and allows for nesting of loops.

Controllers in Sequence Players

The realtime controllers that the Sequence Player responds to are (control numbers in parenthesis): pan (10), volume (7), priority (16), sustain (64), and reverb amount (91). Note that because only one effects bus is supported, reverb amount is used to control effect amount no matter what the effect is.

The compact sequence player also uses controllers 102, 103, 104, and 105 for creating loops. Details of this are discussed in the chapter "Using the Audio Tools."

115731389

The Synthesis Driver

The Synthesis Driver is the Audio Library object used by the Sound Player, the Sequence Player, and application-specific players to create Audio Command Lists, which are passed to the Audio Microcode. This section defines various API calls which can be used by application programmers who want to create their own Players.

Programmers who use the Sequence Player and Sound Player need only be familiar with the initialization of the driver, the alAudioFrame() function that creates audio Command Lists, and the mechanism by which the Synthesis Driver satisfies the need for sound data.

Initializing the Driver

The Synthesis driver needs to be initialized in order to be used. This is accomplished by calling alSynNew() with a configuration structure that specifies the number of virtual voices, physical voices, and effects busses to instantiate. The configuration structure also provides information regarding the Audio DMA callback routines, the Audio Heap, FXType and the audio playback rate to use. (Audio DMA callbacks are discussed later in this chapter.)

Note: The alInit() call will call alSynNew().

The configuration also specifies a callback procedure pointer of type `ALDMANew`, which is used by the synthesis driver initialization procedure to set up callbacks for sound data requests. The procedure specified in the configuration structure is called once during initialization for every physical voice that is instantiated. The Synthesis Driver expects the procedure to return another procedure pointer that defines a callback of type `ALDMAproc`, and a pointer to some state information that can be used in various ways to manage sound data requests.

Note: Only one driver may be instantiated at any given time.

Building and Executing Command Lists

The main function of the Synthesis Driver is to build Audio Command Lists, which are executed by the microcode to synthesize audio. Command lists are built in frames. A frame is a number of samples—usually something close to the number of samples required to fill a complete video frame time at the regular video frame rate (e.g. 30 or 60 Hz).

From an application, the Command List (to synthesize a number of audio samples) is built by making a call to alAudioFrame(). Parameters for this call define the number of samples (which must be a multiple of 16), a physical address of an output buffer where the Microcode will put the audio samples, and a pointer to an array that can be used to store the Command List.

During the construction of the Command List, the Synthesis Driver makes callbacks to its clients (the players) to process the various events that determine the parameters and timing of the playback of sound effects and sequences.

The Driver also makes callbacks to the defined `ALDMAproc` routine with requests for sound data (see below).

To execute an audio Command List, it is first put in `OSTask` structure and then passed to the microcode with a call to osSpTaskStart(). The `OSTask` structure specifies pointers to microcode and data along with the Command List which allows the RCP to execute.

Synthesis Driver Sound Data Callbacks

The application is responsible for making sure that the required sound data is located in RAM before the command list is executed by the audio microcode. The application programmer has the freedom to load complete compressed sounds from the ROM before playback, or, as is more likely, to initiate DMAs from ROM to RAM in response to callbacks from the Synthesis Driver. Initiating DMA's in response to callbacks allows the application to only load the portion of the sound needed, and thus greatly reduce the RAM needed for audio.

The Audio DMA callback routines are initialized when alInit is called. The synthesizer configuration structure must contain a pointer to a routine for initializing the Audio DMA's. This routine will be called once for each physical voice. Typically this routine will initialize any state variables, and then must return a pointer to the ALDMAproc.

The `ALDMAproc` procedure is called by each physical voice during the construction of the command list when compressed sound data is required. The call specifies the required data address, the length, and the state pointer, and it expects to receive a physical memory address where the data can be (or at least will be) found in memory.

The example applications (playseq, and simple) provide examples of how these callback routines can be implemented.

Assigning Players to the Driver

In order to make calls to the driver interface, you must first make your player known to the driver. This is accomplished with the alSynAddPlayer() call. For more information on writing your own player, see the section "Writing Your Own Player".

Note: Both the Sequence Player and the Sound Player add themselves to the driver when they are initialized by calling alSynAddPlayer(). If you are not creating your own players you should not need to call alSynAddPlayer.

Allocating and Controlling Voices

The Synthesis driver manages two types of voices: virtual voices and physical voices.

Virtual voices are described by the ALVoice structure, and represent the voice from the player's perspective. In order to play a wavetable, players must allocate a virtual voice on which to play it. This is accomplished with the alSynAllocVoice() call. The voice configuration structure allows you to specify the voice priority and bus assignment. The number of virtual voices available is established when the driver is initialized, and you may specify more virtual voices than you have resources to play. There is no benefit to specifying more physical voices than virtual voices since the player will have no way to use them.

1151769

Physical voices represent the actual sound processing modules available to the driver. They consist of an ADPCM decompressor, a pitch shifter, and a gain unit. The ADPCM decompressor converts mono ADPCM compressed (approximately 4:1) wavetables to mono 16-bit raw format. The pitch shifter resamples the resulting data (up one octave, down any number of octaves) to the desired pitch. The gain unit then applies a volume envelope, a pan value, and mixes the (stereo) output into the master bus and an effect bus at gains specified by the wet/dry parameters associated with the voice.

The driver maps virtual voices to physical voices based on virtual voice priority. If there are more active virtual voices than available physical voices, the driver allocates the physical voices to the highest priority virtual voices. The driver may "steal" a physical voice from a virtual voice if a higher priority virtual voice is allocated.

Note: To prevent a voice from being stolen, you can set the voice priority to the highest priority with alSynSetPriority().

After you allocate a virtual voice, you can use it to play a wavetable with the alSynStartVoice() call. You can stop the playback with the alSynStopVoice() call.

Once you start a voice, you can control pitch, volume, and panning and effect mix with the appropriate calls listed in the section titled "Summary of Driver Functions".

Effects and Effect Busses

Each voice can be assigned to one effects bus. Each effects bus can contain any number of effects units (up to the limit imposed by the processing resources). The number of busses and effects units are specified in the driver configuration structure and are established at initialization time.

Note: The Audio Library currently only supports one effects bus. Future version may support multiple busses.

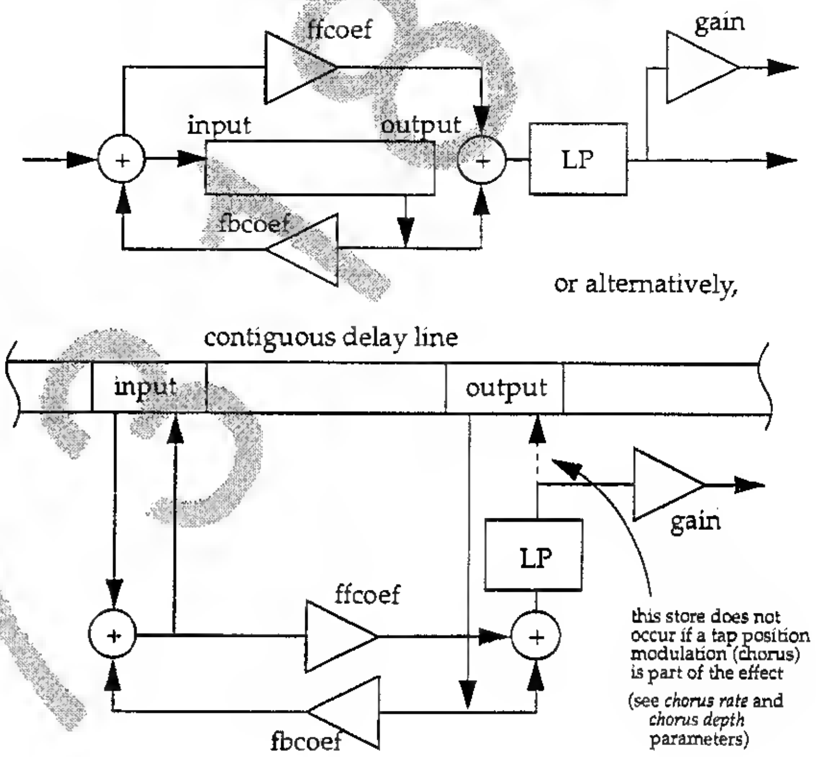
Creating Your Own Effects

The Nintendo64 uses a general purpose effects implementation that manipulates data in a single delay line. A small number of default configurations have been supplied (see libaudio.h), but applications developers can also specify there own custom reverb and chorus/flange style effects.

The way in which the data is manipulated is defined by a set of parameters specified in blocks where each block represents a single effects primitive. An effect is constructed by attaching an arbritrary number of effects primitives to a *single* delay line. There is one and only one input to this delay line which is the sum (slightly attenuated to minimize overflow) of the left and right effects send busses. The contribution of a voice to this bus can be specified by a call to alSynSetFXMix. This delay line is then operated on by the effect specified in the the fxType field of the synthesizer configuration structure. The delay memory will be allocated from the audio heap by a call to `alInit`, so the application must be sure that the audio heap is big enough to contain the delay memory and it's associated effects primitive stuctures. The parameters for each primitive in the effect are specified in an array which is passed to the audio initialization code. Each primitive consists of an input offset, an output offset, coefficients specifying output contribution to input and input contribution to output, chorus rate and depth parameters which control modulation of the output offset, a DC normalized (unity gain at DC) single pole low-pass filter, and finally, an output gain specifying how much of this primitives output is to be contributed to the final effect output.

The particular combination of values in each of the parameters for a primitive specifies the function of that primitive as a whole within the effect. For example, if the ffcoef and fbcoef are the same except for a sign change, that primitive will be an all pass; if ffcoef and fbcoef are different, or one or the other is zero, the primitive will be a filter of some kind. If both ffcoef and fbcoef are zero, the primitive will be pure delay only, possibly modulated and low pass filtered.

Figure 19-2 Effects Primitives



this store does not occur if a tap position modulation (chorus) is part of the effect

(see *chorus rate* and *chorus depth* parameters)

The function of the effects primitives can be thought of in two ways, the first of which is as an individual signal processing block. The effect as a whole would then be thought of as a set of concatenated and/or nested primitives arranged to produce the overall desired effect. The second way of conceptualizing the primitive is the way it is actually implemented, which is to say, as an operator on a single longer delay line shared with all the other primitives. Both conceptualizations are illustrated in figure 13-2. By careful selection of the effects parameters, a large class of cascaded/nested all-pass and comb filter based effects can be created. (For a more detailed description of this class of effects, see Bill Gardner's MIT masters thesis, "The Virtual Acoustic Room", section 4.6, available from

1157389

http://sound.media.mit.edu/papers.html, and his Macintosh "Reverb" program and documentation in same location).

Builders of custom effects will also discover that the effect specification controls not only the nature of the effect, but the processing resources consumed by the effect. Only those functions which are driven by non-zero parameters actually generate any audio command operations in the RCP. This gives application developers a great degree of flexibility in defining an effect that is appropriate both in terms of sonic quality and efficiency. If a developer wishes to use one of the pre-defined effects, they need only specify that effect in the fxType field of the synthesizer configuration structure. If, on the other hand, they wish to build their own effect, they would specify an fxType of AL_FX_CUSTOM, and then allocate and fill in the fields for the primitives. See the PR/apps/playseq source for one example of how to use this capability to build a complex effect.

To create a custom effect, an application specifies the number of sections, the overall length of the delay memory used by the total effect, and then the input and output addresses, feedforward and feedback coefficients, gain, chorus rate and depth, and low-pass coefficient for each section. Following is a brief explanation of the significance of each parameter and what processing actually takes place as a result of it's inclusion. Although parameters are interpreted in different ways, they are all stored in signed 32-bit numbers.

Parameter Description

The following two parameters are specified only once for the entire effect:

sections: this parameter specifies the total number of sections in the effect. A section is one primitive and it's associated parameters.

length: this parameter specifies the total length of delay memory used by the effect, and must be a multiple of 8 bytes. Since data is processed in blocks, this parameter should be greater than or equal to the largest *output* offset parameter PLUS the length of a processing buffer. This length is defined to be 160 samples, or 320 bytes. If the last section of the effect has a non-zero chorus rate parameter which corresponds to a slow modulation rate, and a deep modulation depth (> 1 semitone), the total delay length may need to be larger depending on the rate and depth of the chorus.

1151399

The rest of these parameters constitute one processing element, so there must be one set of these parameters for each section specified by the *sections* parameter.

The following two address parameters must be positive and must be on 8 bytes (or 4 sample) boundaries. The application playseq.c shows an easy way to specify addresses in the convenient unit of milliseconds which are properly aligned.

input: this parameter specifies the address of the input of this section of the effect. This address must be on a 4 sample (or 8 byte) boundary.

output: this parameter specifies the address of the output of this section of the effect. This address must be on a 4 sample (or 8 byte) boundary.

The following three parameters, along with the *lpfilt coef* parameter, are interpreted as signed 16-bit fractional fixed point values. The upper sixteen bits should be sign extended:

fbcoef: this parameter specifies the coefficient of the feedback portion of the section. If this parameter is zero, no action takes place.

ffcoef: this parameter specifies the coefficient of the feedforward portion of the section. If this parameter is zero, no action takes place. If the *chorus rate* parameter is non-zero, because it is not possible to store the loaded output back into the delay line since it is not the same length), the *ffcoef* parameter controls how much of the input to add to the interpolated output allowing flange type effects.

gain: this parameter specifies how much of this primitives output to contribute to the total effect output, and can be thought of as a 'tap' value. If zero, no multiply is performed. Note that at least one section of the effect must have a non-zero gain value for the effect to be heard. If no section of an effect has a non-zero gain value, then no effect output will be heard.

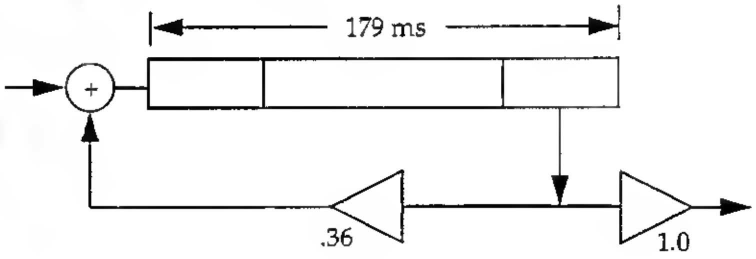
chorus rate: this parameter specifies the modulation frequency of the output tap position of the delay line, i.e., how quickly the tap position will be modulated. The value of this parameter is (frequency/sample rate)*2^25. For example, a modulation frequency of .5Hz at a synthesizer sample rate of 44.1kHz would be (.5/44100)*33,554,432 = 380 *chorus depth*: this parameter specifies the modulation depth, or pitch change, of the effect. The parameter is specified approximately in hundredths of a cent. So a modulation depth of +/-25 cents, or a quarter of a semitone, would be 2500. The approximation to cents is good over the range useful for musical chorusing and flanging, i.e., less than a few semitones. The error at 1 semitone (100 cents) is about 3 cents and at 3 semitones is about 30 cents. If you wish to know the "exact" value (in cents) of the modulation depth , use the following equation:

$$cents = \left|\frac{1200}{\ln(2)}\ln\left(1 - \frac{chorusdepth}{120,000/\ln(2)}\right)\right|$$

lpfilt coef: this parameter specifies the single pole low-pass filter coefficient. The derivation of this value as a function of frequency and sample rate can be found in numerous signal processing texts, and is left as an exercise to the reader (doncha hate that). Generate a table once and forget about it. Only positive values will actually be low-pass. Negative values will generate DC normalized boost at high frequencies causing possible overflow.

Armed with this knowledge about primitive parameters, let's look at some example effects:

Figure 19-3 A simple echo effect



1153189

1157399

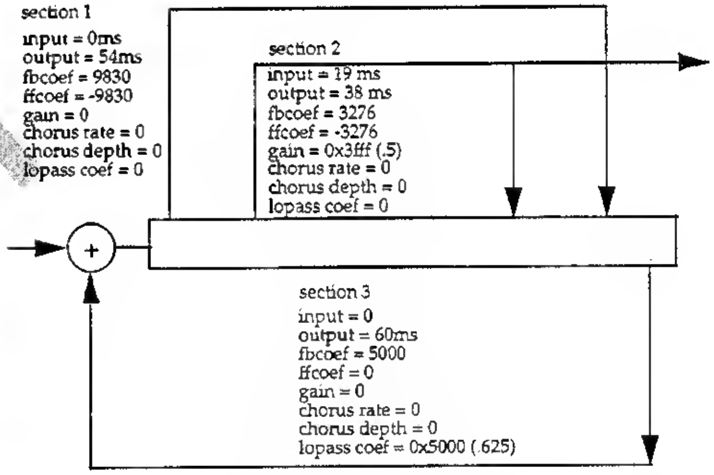
The effect in figure 13-3, which is a simple echo effect, and can be selected using AL_FX_ECHO, would be implemented using the following parameters:

```
#define ms *(((s32)((f32)44.1))&~0x7)
param[0] = 1; /*the number of sections in this effect */
param[1] = 200 ms; /* total allocated memory */
param[2] = 0; /* input is beginning of delay line */
param[3] = 179 ms; /* output location on delay line */
param[4] = 12000; /* fbcoef of .36 */
param[5] = 0; /* no feedforward coefficient */
param[6] = 0x7fff; /* full gain 1.0 - 1/2^15 */
param[7] = 0; /* no chorus rate */
param[8] = 0; /* no chorus depth */
param[9] = 0; /* no low-pass filter */
```

This is, in fact, the echo effect implemented when AL_FX_ECHO is specified in the fxType field of the synthesizer configuration structure.

Let's try something a little more interesting:

Figure 19-4 A nested all-pass inside a comb effect


section 1
input = 0ms
output = 54ms
fbcoef = 9830
ffcoef = -9830
gain = 0
chorus rate = 0
chorus depth = 0
lopass coef = 0
section 2
input = 19 ms
output = 38 ms
fbcoef = 3276
ffcoef = -3276
gain = 0x3fff (.5)
chorus rate = 0
chorus depth = 0
lopass coef = 0
+
section 3
input = 0
output = 60ms
fbcoef = 5000
ffcoef = 0
gain = 0
chorus rate = 0
chorus depth = 0
lopass coef = 0x5000 (.625)

In Fig 13-4, we have used the more compact Gardner-style notation. Note that section 2 is "nested" inside section 1.This effect which is the

AL_FX_SMALLROOM effect, would be specified using the following parameters:

```
param[0] = 3; /*the number of sections in this effect */
param[1] = 100 ms; /* total allocated memory */
/* SECTION 1 */
param[2] = 0; /* input */
param[3] = 54ms; /* output */
param[4] = 9830; /* fbcoef */
param[5] = -9830; /* ffcoef */
param[6] = 0; /* no out gain */
param[7] = 0; /* no chorus rate */
param[8] = 0; /* no chorus delay */
param[9] = 0; /* no low-pass filter */
/* SECTION 2*/
param[10] = 19 ms; /* input */
param[11] = 38 ms; /* output */
param[12] = 3276; /* fbcoef */
param[13] = -3276; /* ffcoef */
param[14] = 0x3fff; /* gain */
param[15] = 0; /* chorus rate */
param[16] = 0; /* chorus depth */
param[17] = 0; /* low-pass filter */
/* SECTION 3*/
param[18] = 0; /* input */
param[19] = 60ms; /* output */
param[20] = 5000; /* fbcoef */
param[21] = 0; /* ffcoef */
param[22] = 0; /* gain */
param[23] = 0; /* chorus rate */
param[24] = 0; /* chorus depth */
param[25] = 0x5000; /* low-pass filter */
```

1153189

Summary of Driver Functions

Table 19-5 Synthesizer Functions

Function	Description
alSynNew	Opens and initializes the synthesizer driver.
alSynDelete	NOT IMPLEMENTED
alSynAddPlayer	Adds a client player to the synthesizer.
alSynRemovePlayer	Removes a player from the synthesizer.
alSynAllocVoice	Allocates and returns a synthesizer voice.
alSynFreeVoice	Deallocates a synthesizer voice.
alSynStartVoice	Starts a virtual voice playing.
alSynStartVoiceParams	Starts a virtual voice with the specified parameters.
alSynStopVoice	Stops a virtual voice from playing.
alSynSetVol	Sets the volume for the specified voice.
alSynSetPitch	Sets the pitch for the specified voice.
alSynSetPan	Sets the pan values for the specified voice.
alSynSetFXMix	Sets the wet/dry/effects/mix for the specified voice.
alSynSetPriority	Sets the priority of the specified virtual voice.
alSynGetPriority	Returns the priority of the specified virtual voice.
alSynAllocFx	Allocates a new effect of the specified type to the specified bus.
alSynFreeFx	NOT IMPLEMENTED

11573169

Table 19-5 Synthesizer Functions

Function	Description
alSynGetFXRef	Returns a pointer to the FX structure.
alSynSetFXParam	Currently has no effect.

Writing Your Own Player

A Player is an Audio Library software object that works through the Synthesis Driver to construct audio command lists. Both the Sequence Player and the Sound Player are examples of Players.

A Player operates by signing into the driver and then responding to driver callback with driver API calls, described in the section "The Synthesis Driver" on page 382. The initialization procedure and the callback routine are detailed below.

Initializing the Player

In order for your player to receive driver callbacks and to use the synthesis driver voice functions, you must first add the player as a driver client. This is accomplished with the alSynAddPlayer() call, which takes two arguments: a reference to the synthesis driver, and a reference to the ALPlayer structure that represents the player to be added. A reference to the synthesis driver may be obtained from the Audio Library globals structure alGlobals. The ALPlayer structure contains a reference to the voice handler callback function and a pointer that the player can use.

Example 19-1 Player Initialization

```
typedef struct MyPlayer_s {

    ALPlayer node;

    /*
     * include other player specific state here
```

11573189

```
     */

} MyPlayer;

void playerNew(MyPlayer *p)
{
    /*
     * Initialize any player specific state here
     */

    /*
     * Sign into the synthesis driver so that the next time
     * alAudioFrame is called, it will call the
     * __voiceHandler function.
     */
    p->node.next       = NULL;
    p->node.handler    = __voiceHandler;
    p->node.clientData = p;
    alSynAddPlayer(&alGlobals->drvr, &p->node);
}

void playerDelete(MyPlayer *p)
{
    /*
     * remove this player from the synthesis driver
     */
    alSynRemovePlayer(&alGlobals->drvr, &p->node);
}
```

In the previous example, you'll notice that the player structure contains a reference to __voiceHandler. This field points to a callback procedure, of type ALVoiceHandler, which the driver calls in the process of building the audio command list.

Implementing a Voice Handler

When your application calls alAudioFrame(), the driver iterates through its list of players, calling the player's voice handler functions at the appropriate offset (which translates to time) in the command list.

Typically, the player maintains a time-based list of events which the voice handler parses and translates into driver calls. The voice handler contributes to the construction of the command list by making driver voice calls.

Note: Driver voice calls can be made only from within the voice handler function.

The voice handler returns the time, in microseconds, for the next callback.

Example 19-2 The Voice Handler

```
ALMicroTime __voiceHandler(void *node)
{
    MyPlayer    *p = (MyPlayer *)node;

    /*
     * You can now make calls to the following synthesis
     * driver voice functions
     *
     *         alSynAllocVoice()
     *         alSynFreeVoice()
     *         alSynStartVoice()
     *         alSynStopVoice()
     *         alSynSetVol()
     *         alSynSetPitch()
     *         alSynSetPan()
     *         alSynSetFXMix()
     *         alSynSetPriority()
     *         alSynGetPriority()
     *         alSynSetFXParam()
     */

    return 1000;        /* call back in 1 millisecond */
}
```

1157389

Implementing Vibrato and Tremolo

1157389

Note: A full example of vibrato and tremolo implementation is given in the latest version of the playseq demo. GenMidiBank.inst has examples of how vibrato and tremolo would be set in the bank.

Vibrato and tremolo, are implemented by providing three callback routines; initOsc, updateOsc, and stopOsc. These routines act as the low frequency oscillator (LFO) that is modulated against either pitch or volume. When the sequence player determines that a note uses either vibrato or tremolo, it will call initOsc which will set a current value, and return a delta time specifying how long before it needs to update the value of the oscillator. After the delta time has passed, updateOsc will be called, which will set a current value and return a delta time until the next update. This will continue, until the note stops sounding, and at that time, stopOsc will be called, so that your application can do any necessary cleanup.

What each routine does, and how it does it is largely up to the application. All the sequence player expects is a delta time until the next callback, and a value to use as the current value. In addition the sequence player provides a mechanism for each note to have its own data, and for this data to be passed to subsequent calls of updateOsc.

For vibrato or tremolo to be active, you must set the vibType or tremType of the instrument in the .inst file. A value of zero (the default) in these fields will be interpreted by the sequence player as either vibrato off or tremolo off. Any non-zero value will be considered as on. In addition to the type, the following fields can be used to specify parameters for the oscillator: vibRate, vibDepth, vibDelay, tremRate, tremDepth, tremDelay. These values are eight bit values and can be used in whatever way the oscillator callbacks deem appropriate.

When creating a sequence player, you must pass pointers to your callbacks through the ALSeqpConfig struct. The following code fragment demonstrates how to do this.

```
ALSeqpConfig    seqc;

seqc.maxVoices       = MAX_VOICES;
seqc.maxEvents       = EVT_COUNT;
seqc.maxChannels     = 16;
seqc.heap           = &hp;
seqc.initOsc        = &initOsc;
seqc.updateOsc      = &updateOsc;
seqc.stopOsc        = &stopOsc;

alSeqpNew(seqp, &seqc);
```

The initOsc routine

```
ALMicroTime initOsc(void **oscState, f32 *initVal, u8
    oscType,u8 oscRate, u8 oscDepth, u8 oscDelay);
```

The initOsc routine is the first callback to occur when a note is started, and either the vibType or tremType is non-zero. Vibrato and tremolo are handled separately by the sequence player, so if an instrument has both vibrato and tremolo, two calls will be made, one for each oscillator. When called, initOsc is passed a handle, in which it may store a pointer to a data structure. This pointer will be passed back to subsequent calls of updateOsc and stopOsc. This is optional. The second argument is a pointer to an f32 that must be set with a valid oscillator value. The remaining arguments are the oscType, oscRate, oscDepth, and oscDelay. These values may be used as you wish.

Typically initOsc will allocate enough memory for its data structure, and store a pointer to this memory in the oscState handle. This is optional though, and if your oscillator doesn't have any state information it may not need to do this. After performing any computation that it needs, the initOsc routine returns a delta time, in microseconds, until the first call to updateOsc. If a delta time of zero is returned, the sequence player interprets this as a failure, and will not making any calls to either updateOsc or stopOsc. If the initVal is changed, the new value will be used. If the initVal remains unchanged, vibrato will default to a value of 1.0 and tremolo will default to a value of 127.

If the oscillator is a vibrato oscillator, the return value is multiplied against the unmodulated pitch to determine the modulated pitch. A value of 1.0 will have no effect, a value of 2.0 will raise the pitch one octave, and a value of .5 will lower the pitch one octave. If the oscillator is a tremolo oscillator, the returned f32 should be an integer value between 0 and 127. This value will be multiplied against the unmodulated volume to determine a modulated volume. A value of 127 will be full volume, and a value of 0 will be silent.

The updateOsc routine

```
ALMicroTime updateOsc(void *oscState, f32 *updateVal);
```

The updateOsc routine will be called whenever the delta time returned by either initOsc or the previous updateOsc call has expired. When called, updateOsc is passed the value returned by initOsc in the oscState handle. UpdateOsc should make whatever calculations it needs, set the new oscillator value in updateVal, and return a delta time until the next time updateOsc needs to be called. Valid oscillator values are the same as in the case of initOsc.

The stopOsc routine

```
void stopOsc(void *oscState);
```

The main purpose of the stopOsc routine is to give the application the opportunity to free any memory stored in the oscState. StopOsc is not called until the note has completely finished processing. Even if your routine does nothing, you should still have a stopOsc routine if you have an initOsc routine.

11573169

11573189

Chapter 20

Audio Tools

1157389

This chapter describes the various audio tools for the Nintendo 64. These include: an instrument compiler, which can be used to prepare banks of sounds and control information used by the sequence player and the sound player; a set of tools to compress and decompress sound data for the Nintendo 64 ADPCM format; and tools for converting and printing MIDI files.

The Instrument Compiler: ic

The Nintendo 64 Audio Library synthesizes audio from MIDI events using information contained in the .ctl and .tbl data files. These files, along with the .sym file, are known collectively as Bank files, and are created by the "ic" tool.

The .tbl file contains the ADPCM compressed audio wavetable data.

The .ctl file contains information about how the wavetables are to be synthesized. It includes information about the wavetable's envelope, pan position, pitch, mapping to MIDI note numbers, and velocity values. For more information about the format of the .ctl file, see the section "Bank Files" in Chapter 15

The .sym file contains the bank file's symbol information, and is used mainly for development and debugging. It is used only by the audio bank tools, not by the Audio Library.

Note: ic can also be used to collect sound effects into a single bank structure for inclusion in the ROM. In this case some of the features of the Bank format are not used (for example, Keymaps and Instrument parameters).

Invoking ic

Invoke ic by entering this command:

```
ic [-v] -o <output file prefix> <source file>
```

11573189

Table 20-1 ic Command Line Options

Command Line Option	Function
-v	Turns on verbose mode, which causes the compiler to produce a quantity of largely useless information.
-o <output file prefix>	Specifies the prefix for the .ctl, .tbl, and .sym files created by the compiler.
<source file>	The name of the file containing the source code for the banks of instruments.

Writing ic Source Files

Instrument Compiler source files consist of C-like definitions for the collection of objects that make up the Bank. There are objects to represent banks, instruments, sounds, keymaps, and envelopes. Each of these objects is detailed below.

The Bank Object

A bank object, denoted by the keyword "bank," contains an array of instruments, a sample rate specification, and an optional default percussion instrument. In the example below, the bank defined as "GenMidiBank" contains one instrument, called "GrandPiano," at instrument location 0. It is intended to operate at 44.1 kHz.

```
bank GenMidiBank
{
sampleRate = 44100;
program [0] = GrandPiano;
}
```

Note: The General MIDI 1.0 Specification specifies that MIDI channel 10 is the default drum or percussion channel. As a result, many General MIDI sequences do not contain program change messages for channel 10. You can specify the default instrument (program) for channel 10 as follows:

```
bank GenMidiBank
{
sampleRate = 44100;
percussionDefault = Standard_Kit;
program [0] = GrandPiano;
}
```

The Sequence Player sets the default instrument for channel 10 messages to be "Standard_Kit."

The Instrument Object

The instrument object, referenced by the bank object, contains the overall volume and pan for the instrument as well as the list of sounds that make up the instrument.

In the example below, the "GrandPiano" instrument contains eight sounds: "GrandPiano00", "GrandPiano01", "GrandPiano02", "GrandPiano02", "GrandPiano03", "GrandPiano04", "GrandPiano05", "GrandPiano06", and "GrandPiano07".

The overall instrument volume is 127, or full volume, and is panned to the position 64, which is center.

```
instrument GrandPiano
{
    volume = 127;
    pan    = 64;

    sound [0]  = GrandPiano00;
    sound [1]  = GrandPiano01;
    sound [2]  = GrandPiano02;
    sound [3]  = GrandPiano03;
    sound [4]  = GrandPiano04;
    sound [5]  = GrandPiano05;
    sound [6]  = GrandPiano06;
    sound [7]  = GrandPiano07;
}
```

11573189

The Sound Object

The sound object specifies the volume and pan, keyboard mapping, and envelope for the sound. It also specifies the AIFF-C sound file containing the ADPCM compressed wavetable data. A description of the AIFF-C format expected by ic (which is generated by the ADPCM encoding tools) is given in the section titled "ADPCM AIFC Format" in Chapter 21.

Note: The Sequence Player multiplies the instrument volume with the sound volume to get the overall volume. It *adds* the instrument pan with the sequence pan to get the sound's overall pan.

In the example below, the GrandPiano00 sound specifies that the wavetable data is to come from the file *../sounds/GMPiano_C2.18k.aifc*. It is to be panned center (64) at full volume (127) and arranged on the keyboard according to the map specified in `piano00key` with the envelope specified in `GrandPianoEnv`.

```
sound GrandPiano00
{
    use ("../sounds/GMPiano_C2.18k.aifc");
    pan    = 64;
    volume = 127;
    keymap = piano00key;
    envelope = GrandPianoEnv;
}
```

Keymaps and envelopes are described in the following sections.

Note: When using banks to collect sound effects, the keymap entry is not necessary.

The Keymap Object

The keymap object, referenced by the sound object, specifies the range of MIDI velocities and key numbers that the sound is intended to cover. It is used by the Sequence Player to determine which sound to map to a given MIDI note number, and at what pitch ratio to play the sound.

In the example below, `piano00key` specifies a MIDI Note On message with a velocity between 0 and 127 and a note number between 0 and 43

In this example, the keyBase is 41, so a MIDI Note on message for key 41 triggers the sound that references this keymap at unity pitch. A MIDI Note On message for key 42 triggers the same sound, but shifted up a half step in pitch.

Note: You can set the keyBase value outside the range of keyMin to keyMax. This is useful if you want to critically resample a wavetable to conserve ROM space. You could, for instance, resample a wavetable from 44.1 kHz to 22.05 kHz and adjust the keyBase up an octave to compensate. Remember, however, that quality degrades at larger pitch shift ratios.

The detune parameter indicates the number of cents that is to be added to the default tuning. A half step is equal to 100 cents.

```
keymap piano00key
{
    velocityMin = 0;
    velocityMax = 127;
    keyMin      = 0;
    keyMax      = 43;
    keyBase     = 41;
    detune      = 0;
}
```

The Envelope Object

The envelope object specifies the attack-decay-sustain-release (ADSR) envelope, or volume contour, for a sound. Volumes are specified in the range of 0 to 127, and the times are specified in microseconds.

In the example below, the sound's envelopes would ramp from 0 to 127 in 0 microseconds, decay to 0 in 400 milliseconds, wait for a MIDI Note Off, and then release to 0 in 200 milliseconds. The decay portion of the envelope decays to zero. For many acoustic instruments, especially percussion instruments, this gives the most realistic envelope.

1151399

Note: The Sound Player uses envelopes in a slightly different way. See Chapter 19 for details.

A Complete Example

The following example, taken from the General MIDI bank that is shipped with the development software, defines a bank with one instrument, the Grand Piano.

```
envelope GrandPianoEnv
{
    attackTime= 0;
    attackVolume= 127;
    decayTime= 4000000;
    decayVolume= 0;
    releaseTime= 200000;
    releaseVolume= 0;
}

keymap piano00key
{
    velocityMin = 0;
    velocityMax = 127;
    keyMin      = 0;
    keyMax      = 41;
    keyBase     = 51;
    detune      = 0;
}

sound GrandPiano00
{
    use ("../sounds/GMPiano_C2.18k.aifc");
    pan    = 64;
    volume = 127;
    keymap = piano00key;
    envelope = GrandPianoEnv;
}

keymap piano01key
{
    velocityMin = 0;
    velocityMax = 127;
```

11573189

```
    keyMin      = 42;
    keyMax      = 49;
    keyBase     = 63;
    detune      = 0;
}

sound GrandPiano01
{
    use ("../sounds/GMPiano_Bb2.16k.aifc”);
    pan    = 64;
    volume = 127;
    keymap = piano01key;
    envelope = GrandPianoEnv;
}

keymap piano02key
{
    velocityMin = 0;
    velocityMax = 127;
    keyMin      = 50;
    keyMax      = 57;
    keyBase     = 67;
    detune      = 0;
}

sound GrandPiano02
{
    use ("../sounds/GMPiano_F3.19k.aifc”);
    pan    = 64;
    volume = 127;
    keymap = piano02key;
    envelope = GrandPianoEnv;
}

keymap piano03key
{
    velocityMin = 0;
    velocityMax = 127;
    keyMin      = 58;
    keyMax      = 63;
    keyBase     = 72;
    detune      = 0;
}

sound GrandPiano03
```

11573189

```
{
    use ("../sounds/GMPiano_C4.22k.aifc");
    pan    = 64;
    volume = 127;
    keymap = piano03key;
    envelope = GrandPianoEnv;
}

keymap piano04key
{
    velocityMin = 0;
    velocityMax = 127;
    keyMin      = 64;
    keyMax      = 69;
    keyBase     = 79;
    detune      = 0;
}

sound GrandPiano04
{
    use ("../sounds/GMPiano_G4.22k.aifc");
    pan    = 64;
    volume = 127;
    keymap = piano04key;
    envelope = GrandPianoEnv;
}

keymap piano05key
{
    velocityMin = 0;
    velocityMax = 127;
    keyMin      = 70;
    keyMax      = 75;
    keyBase     = 84;
    detune      = 0;
}

sound GrandPiano05
{
    use ("../sounds/GMPiano_C5.22k.aifc");
    pan    = 64;
    volume = 127;
    keymap = piano05key;
    envelope = GrandPianoEnv;
}
```

11573189

```
keymap piano06key
{
    velocityMin = 0;
    velocityMax = 127;
    keyMin      = 76;
    keyMax      = 81;
    keyBase     = 91;
    detune      = 0;
}

sound GrandPiano06
{
    use ("../sounds/GMPiano_G5.22k.aifc");
    pan    = 64;
    volume = 127;
    keymap = piano06key;
    envelope = GrandPianoEnv;
}

keymap piano07key
{
    velocityMin = 0;
    velocityMax = 127;
    keyMin      = 82;
    keyMax      = 111;
    keyBase     = 99;
    detune      = 0;
}

sound GrandPiano07
{
    use ("../sounds/GMPiano_C6.18k.aifc");
    pan    = 64;
    volume = 127;
    keymap = piano07key;
    envelope = GrandPianoEnv;
}

instrument GrandPiano
{
    volume = 127;
    pan    = 64;

    sound [0]  = GrandPiano00;
```

11573189

```
    sound [1]  = GrandPiano01;
    sound [2]  = GrandPiano02;
    sound [3]  = GrandPiano03;
    sound [4]  = GrandPiano04;
    sound [5]  = GrandPiano05;
    sound [6]  = GrandPiano06;
    sound [7]  = GrandPiano07;
}

bank GenMidiBank
{
sampleRate = 44100;
program [0] = GrandPiano;
}
```

11573189

The ADPCM Tools: tabledesign, vadpcm_enc, vadpcm_dec

The ic tool requires wavetables to be compressed in ADPCM format before they are included in a sound bank. ADPCM compression is accomplished using the tabledesign, vadpcm_enc, and vadpcm_dec tools. These tools are described below.

Note: The format described is used only as an interchange format between the compression tools and the instrument compiler. It is not used to store compressed sound data on the ROM.

tabledesign

tabledesign reads an AIFC or AIFF sound file and produces a codebook (written to standard output), which is used by the ADPCM encoder. The codebook is a table of prediction coefficients which the coder selects from to optimize sound quality. The procedure used to design the codebooks is based on an adaptive clustering algorithm.

Invoking tabledesign

```
tabledesign [-s book_size] [-f frame_size]
[-i refine_iter] aifcfile
```

1151789

Command-line options are described in Table 14-2.

Table 20-2tabledesign Command Line Options

Command Line Option	**Function**
-s <value>	Value is the base 2 log of the number of entries in the table. Currently up to 8 entries are supported, so the value can range from 0 to 3. The default value for this parameter is 2, giving 4 entries. This seems to be adequate for most sounds.
-f <value>	Value is the size of the frames (in samples) used to estimate predictors. Since the ADPCM encoder operates on frames of 16 samples, this number should be a multiple of 16. The default value is 16. The main benefit of increasing the frame size is that design time is reduced.
-i <value>	Value is the number of iterations used in the refinement step of the clustering algorithm. The default value is 2. Increasing this parameter increases design time, with some possible improvement in quality. The default is adequate for most sounds.

vadpcm_enc

vadpcm_enc encodes AIFC or AIFF sound files and produces a compressed binary file, which is used by ic to prepare banks of sounds. The encoding algorithm is based on a switched ADPCM algorithm which uses a codebook to define a table of prediction coefficients. Coefficients from the table are selected adaptively during encoding to give the best sound quality. The Nintendo 64 compressed sound format currently supports a single loop point, which should be defined in the input file's Instrument Chunk. The codebook and loop-point definitions are embedded in the final output file.

11573189

Invoking vadpcm_enc

The vadpcm_enc tool is invoked as follows:

```
vadpcm_enc -c codebook [-t] [-l minLoopLength]
aifcFile codedFile
```

Table 20-3vadpcm_enc Command Line Options

Command Line Option	Function
-c <filename>	Define a file that contains the prediction coefficient codebook constructed by tabledesign(1).
-t	Truncate the encoded file after the loop end point. The portion of the sound after the loop end-point is never used in audio playback.
-l <value>	Set the minimum loop length in the encoded file (see Note below).

Note: The efficiency of wavetable synthesis is dependent on the length of loops. Longer loop lengths can be synthesized more efficiently. A minimum loop length can be set in the ADPCM encoder. The currently defined default minimum loop length is 800 samples. This default length can be changed (see above), with the absolute minimum being 16 samples. Loops shorter than the minimum loop length are repeated until the total loop length is larger than the minimum length. If possible loops should be longer than a single audio frame which is equal to the (SampleRate)/(FrameRate).

vadpcm_dec

vadpcm_dec decodes a sound file that has been encoded in the Nintendo 64 ADPCM format using vadpcm_enc, and writes it to standard output as raw mono 16-bit samples.

Invoking vadpcm_dec

The vadpcm_dec tool is invoked as follows:

```
vadpcm_dec [-l] codedfile
```

Table 20-4vadpcm_dec Command Line Options

Command Line Option	**Function**
-l	If the sound has a loop, play the loop repeatedly until a key is pressed on the standard input.

115731389

11573189

The MIDI File Tools: midicvt, midiprint & midicomp

midicvt

The Audio Library plays only Type 0 Standard MIDI files. You can use midicvt to convert from Type1 (which are generally output by most MIDI sequencers) to Type0.

Invoking midicvt

midicvt is invoked as follows:

```
midicvt [-v] [-s] <input file> <output file>
```

Table 20-5midicvt Command Line Options

Command Line Option	Function
-v	turns on verbose mode
-s	strips out any messages that are not used by the Audio Library. These include text messages and system exclusives.
input file	the name of a Type 0 or Type 1 Standard MIDI file.
output file	the name for the Type 0 output file.

midiprint

The midiprint tool prints a text listing of the time-based MIDI events in a Type 0 or Type 1 Standard MIDI file.

Invoking midiprint

```
midiprint [-v] -o <output file> <input file>
```

Table 20-6midiprint Command Line Options

Command Line Option	Function
-v	verbose mode.
-o <output file>	the optional output file for the MIDI event text.
<input file>	the name of the Type 0 or Type 1 Standard MIDI file to list.

midicomp

The midicomp tool is used to compress midi files of either Type 0 or Type 1 to a format recognized by the compact sequence player.

Invoking midicomp

midicomp is invoked as follows:

```
midicomp <input file> <output file>
```

Table 20-7midicomp Command Line Options

Command Line Option	Function
<input file>	the name of the Type 0 or Type 1 Standard MIDI file to compress.
<output file>	the name to use for the output file.

Making files that will compact better.

Different midi files will be compressed by different percentages, based on the content of the files. All files (except very small files) should be compressed at least somewhat. Because midicomp achieves compression by recognizing patterns and then compressing these, the greatest amounts of compression occur when the files are repetitive. Patterns and sections created in a sequencer using cut and paste are the ones most likely to be compressed.

115731899

Midi Receiving with Midi Daemon: midiDmon

Midi Daemon is no longer supported. All functionality from *Midi Daemon* is now incorporated into *Instrument Editor*.

115731890

Instrument Editor

The tool *Instrument Editor* provides three primary uses. First, as an editor, it allows realtime editing and auditioning of instrument banks and effects. Second, as a player, it allows external MIDI devices to playback MIDI on the Nintendo 64 Development Hardware. Third, as a profiler, it profiles and measures audio resources that are being used during playback. With its support for MIDI playback, the ie tool is intended to replace the functionality of the *Midi Daemon* tool.

Instrument Editor is invoked with the command:

```
ie [-b <.inst file>] [-c <.cnfg file>] [-v]
```

Table 20-8 ie Command Line Options

Command Line Option	Function
-b <.inst file>	specifies the name of the instrument bank file to open in the editor. If this option is not used, the editor opens with a new .inst file.
-c <.cnfg file>	specifies the name of the configuration file used to configure the N64 Audio Library used by ie.
-v	turns on verbose mode. (for debugging.)

Editor

The editor portion of the ie tool is a simple application for editing .inst files as well as effects. A Nintendo 64 development board does not have to be present to open and edit .inst files. However, you will not be able to audition your changes without the Nintendo 64.

Bank Editing

The ie tool can read, write, and edit .inst files. .inst files contain a description of a Nintendo 64 bank which can be compiled into actual Nintendo 64 bank files with ic, the instrument compiler tool. The .inst bank description is

1157319

made up of several components such as instruments, sounds, envelopes, etc. Each of these bank components, or assets, have one or more parameters associated with it. For example, an instrument asset as volume, pan, and bend range parameters associated with it among others. Assets can also reference each other in a sort of parent-child relationship. For instance, bank assets reference instruments assets so instruments are children of a bank. Similarly, instrument assets reference sounds assets so sounds are children of an instrument. Furthermore, if a child asset is never referenced by another asset (ie. it has no parent), it is called an orphan. So if an envelope asset is never used by a sound asset, the envelope is an orphan and can be deleted from the .inst file without affecting the bank.

Viewing Assets

The editor displays all these bank assets and supports viewing and editing the parent-child relationships within a bank. The editor's view contains several folders for each type of bank asset. Each folder contains a list of all the assets of the given type. For example, to view a bank's instruments, simply select the instrument's folder tab to open up the instrument folder. The folder contains a list of all the names of the instruments as well as columns for each of an instrument's parameters, such as volume, pan, priority, and bend range. Each asset also contains an icon column which helps identify the type of asset.

Editing Assets

To edit the value of an asset's parameters, simply click on the corresponding column to activate the default editing for the parameter. Names are always text edited. Numbers can be scrolled up or down to increase or decrease their value. References to other child assets are edited with popup menus. However, all assets can be text edited by clicking on them with the "Alt" key held down. This pops up a text edit field which can be moved around from field to field using the arrow keys and the "Alt" key. (Without the Alt key, the arrow keys move the cursor within the text field.) Values won't be accepted if the value is out of range or is illegal. Use the "ESC" key to cancel any text editing. Note that some fields cannot be edited (eg. a wavetable's sample rate) and only display information. Icon fields are used for a variety of purposes such as asset selection, asset audition, and others. Integer fields can be double-clicked to quickly set the value to a preset default value.

Viewing and Editing Children

Some of the assets contain a "#" column. This column displays the number of children that the asset has. If the asset has one or more children, double-clicking on the "#" column will open up the parent and display its children. Since the children have different parameters than the parent, only the common fields such as the name field are displayed for children. Double-clicking the "#" column again will close the asset. The "#" field can be edited by clicking on the field. This will bring up a popup menu showing a list of assets that are currently not children of the selected asset. Choosing one of these assets will add it to the parent's list of children. Double-clicking on the icon of a child, will automatically open up the children's folder for editing of their parameters. For example, double-clicking an instrument's sound will open up the sound folder for editing. Likewise, double-clicking a sound's envelope will open up the envelope folder for editing.

Auditioning Assets

In order to audition assets, the current bank being edited must be "valid" and must be "online" on the Nintendo 64. For a description of what it means for a bank to be valid and online, see the Nintendo 64 Playback section. When a bank is online, bank assets can be auditioned by clicking on their icon. Pressing the button down sends a MIDI note on event. Releasing the button sends a MIDI note off event. This makes it easy to audition the sustain portion of a sound. Currently, auditioning instrument assets will always play a C4 note. Auditioning sounds, keymaps, envelopes, and wavetables will play the asset's parent instrument at the sound's key base. Note that if the keymaps for an instrument's sounds are not specified and ordered properly, an auditioned asset may not get mapped to the correct sound. This is a potential source of confusion when auditioning assets so make sure that the auditioned sound's keymap is correct and complete before auditioning.

The File Menu

The file menu contains commands for opening, closing, and saving .inst files. The "Open" command brings up a dialog for selecting a .inst file to edit. Only one .inst file can be open at a time so choosing "Open" while another .inst file is currently open will first close the file before opening a new one. The "Close" command removes all bank assets and allows a new file to be edited. The "Save" and "Save As" command write the file to disk.

The Edit Menu

The edit commands are currently not supported.

The Asset Menu

The Asset menu contains commands for inserting and deleting assets. Selecting the insert command will create a new asset and place it at the end of the list. The asset will automatically have default parameter values. To insert an asset in the middle of the list, select the asset where you want the asset to appear and select the insert command. The selected asset will appear below the newly created one. To delete assets, simply select one or more assets and select the delete command. A short cut for creating an asset and adding it to a parent is provided by the "Insert Child" command. This command will insert a new child asset to the selected parent. The "Remove Child" command removes the selected child(ren) from the parent, but does NOT delete them. Choose the "Delete" command to remove and delete them. Finally, the "Import" command allows importing of other .inst files as well as .aiff-c files. This is currently the only way to create wavetable assets.

The Select Menu

The select menu contains useful commands for selecting certain types of assets. The "Select Parents" command will select all the parents of the currently selected asset. This command works only if exactly one asset is selected. For example, if a keymap is selected, the "Select Parents" command will select all the sound assets that use the given keymap and will automatically display the sound folder. The "Select Orphans" commands will select all the folder's assets that do not have any parents. This is useful for determining which assets aren't being used anywhere and which can be deleted.

Effects

The ie tool supports creating, editing, and auditioning effects on the Nintendo 64. Since effects are tightly coupled to the N64 Audio Library, they will only appear for editing if N64 development hardware is present.

Otherwise, only bank components can be edited. If N64 development hardware is present, ie will automatically create five built-in effects for auditioning and editing. These effects are small room, big room, chorus, flange, and echo. In addition to the built-in effects, custom effects can be created from scratch.

Effects Viewing

Similar to banks, effects are made up of two components, the effect asset and the effect section asset. Simple effects may contain only one or two sections, while more complicated effects may contain eight or more sections. Similar to banks, effects are parents to effect section children. As a result, effects can be viewed just like bank assets can be viewed. All effects parameter values are displayed in their native data format (the format that the N64 requires them in) except for the delay fields (length, input, and output). The delay parameters are displayed in milliseconds and must be converted to samples and aligned to an 8 sample boundary before being used to configure a game. (ie does this automatically when it loads an effect for auditioning.)

Effects Editing

Effects and effect sections can be edited just like bank assets. However, there are some special considerations when editing effects.

First, the delay parameters (length, input, output) are displayed and editing in msecs. The N64 requires that these values occur at 8 sample boundaries and that the length is greater than both the input and output delays by about 160 samples (depending on the chorus rate). (See the section on audio effects for a more detailed explanation of the 160 sample restriction). The ie tool automatically enforces the 8 sample boundary rule when it loads the effect on the N64, however it does not enforce the 160 sample rule. Be careful when editing input or output delays so that they do not approach within 160 samples (depending on the chorus rate) of the delay line's length. Normally, if this limit is exceeded, you will hear artifacts in the audio such as clicks and pops.

Secondly, when an effect is "online" (ie. it is loaded into the N64), the effect's length parameter cannot be edited. In addition, you cannot insert or delete sections to an online effect. In order to make these changes to an online effect, you must offline the effect first.

11573989

Thirdly, effect sections can only have one parent. Once it is being used by a parent effect, it will not be available for other effects to use it.

Finally, to use chorus or the low pass filter, you must make sure that the respective parameters are non-zero before loading the effect. The Audio Library will not allocate the required memory to implement chorus or the low pass filter if the parameters are initially zero (this saves unneeded memory).

Effects Auditioning

Initially, no effects are loaded onto the N64. In order to load an effect and make it "online", double-click the desired effect's icon. To offline the effect, double-click it again or double-click another effect. When an effect is placed online, the N64 must be fully reconfigured since the Audio Library must be initialized with an effect. This may take a few seconds since it must reload the entire bank to the N64. Once the effect is online, its icon should appear in red to indicate that it is online. From now on, auditioning bank assets will be played through the effect. Note that the wet/dry amount can be controlled for each MIDI channel by sending an FX1 control message to the channel.

Effects Saving and Restoring

Currently, effect assets can not be saved to disk. This is because there is no standard ".fx" file like there is an ".inst" file for bank assets. However, effects can be restored from disk with a configuration (.cnfg) file. (See the section on the N64 Configuration for a description of the configuration file.) Since the Audio Library treats effects as part the the configuration data you can edit the configuration file to include a custom effect. An effect is defined with the keyword "REVERB_PARAMS" and is followed by a bracketed {...} set of parameters describing the effect and its sections. Below is an example of an effect with 8 sections and a total delay line length of 325 msecs. Note that comments are bracketed by /* ... */.

```
REVERB_PARAMS = {
/*  sections      length*/
      8,            325,
/*                                          chorus chorus fltr*/
/* input  output fbcoef   ffcoef    gain     rate   depth coef*/
```

```
     0,     8,      0,  -9830,    3600,     0,     0,      0,
     8,    12,   9830,  -9830,  0x2b84,     0,     0, 0x5000,
    41,   128,  16384, -16384,  0x11eb,     0,     0,      0,
    45,   103,   8192,  -8192,       0,     0,     0,      0,
   162,   282,  16384, -16384,  0x11eb,     0,     0, 0x6000,
   166,   238,   8192,  -8192,       0,     0,     0,      0,
   238,   268,   8192,  -8192,       0,     0,     0,      0,
     0,   299,  18000,      0,       0,   380,  2000, 0x7000}
```

Nintendo 64 Player and Profiler

When ie is launched, it automatically looks for an N64 development board and if it finds one, it will boot it up with MIDI playback code and profiling code. If it can't find the N64 board or if it fails to boot it up, it will report an error and ie will not be able to audition any instruments or edit effects. In addition, ie will also boot up the gload tool which acts as a print server for any error or debugging messages. This is useful for detecting when an audio library resource has been exceeded. If another gload is running at the time that ie is launched, ie will fail to run.

Nintendo 64 Configuration

The Nintendo 64 Audio Library is configured using default configuration information. This default configuration can be edited either by using the configuration dialog or by specifying a configuration file on the command line when the tool is run. For information on how to use the configuration dialog see the section on the Nintendo 64 Menu. To configure the tool using a configuration file, simply specify the file on the command line. The configuration file should contain reserved words that specify the values of certain configuration parameters, such as output rate or the number of available virtual voices. For an example of a .cnfg file and its reserved words, refer to the file /$ROOT/usr/src/PR/assets/banks/ie.cnfg.

Nintendo 64 MIDI Playback

Once it is up and running, the Nintendo 64 waits for incoming MIDI messages. MIDI messages can be sent from an external MIDI device or from the ie tool itself. In order for the Nintendo 64 code to respond to the MIDI messages, it needs to have a valid bank downloaded to it by ie. When ie is launched with a new file, there is no bank in the editor and the Nintendo 64 will be "offline" which means it does not have a bank installed. The profiling screen on the Nintendo 64 monitor indicates the state of the bank at the top of the screen. As soon as ie has a valid bank in the editor, it will download the bank data and the Nintendo 64 will then be "online" and it will be able to respond to MIDI events. As the bank is edited, it continually checks to see whether the bank is still "valid" and as soon as the bank fails to be valid, it will take the bank offline. The reason for this is simply that the Audio Library requires complete and correct bank data in order to work properly. A bank is determined to be valid if the following conditions are met:

1) a bank asset exists
2) the bank contains at least one instrument
3) the bank's instruments contain at least one sound
4) the bank's sounds must all have keymaps, envelopes, and wavetables

When a bank is online, bank assets can be auditioned from the editor by clicking on their icon. MIDI messages can also be sent from external devices. To use external devices, a MIDI interface must be properly attached to one of the host computer's serial ports and it must be properly configured using the startmidi tool.

Nintendo 64 Profiling

The Nintendo 64 screen displays current readings for various audio resources. These readings are useful to monitor when playing back a sequence targeted for the Nintendo 64 from an external MIDI sequencer. The readings will measure how much of each resource is used in order to playback the sequence. The profiler keeps track of the following resources:

115789

Table 20-9ie Profiled Resources

Profiled Resource	Description
cmds	the number of audio commands used to synthesize a frame of samples. Profiles both current and maximum values.
syn upds	the number of parameter update blocks used by the synthesis driver to store changes in control parameters. The number of available update blocks is specified during the Audio Library configuration. Profiles both current and maximum values.
seq evts	the number of event message blocks used by the sequence player. The number of available message blocks is specified during the Audio Library configuration. Profiles both current and maximum values.
DMAs	the number of DMA requests made during an audio frame. Displays both current and maximum values. The maximum number of DMA requests is specified during the audio system configuration. Profiles both current and maximum values.
DMA bufs	the number of DMA buffers needed during an audio frame. The number of availabe DMA buffers is specified during the audio system configuration. Profiles both current and maximum values.
Vcs	this graph profiles virtual voice usage during playback. Each pixel represents one used virtual voice. The number of available virtual voices is specified during the Audio Library configuration. The maximum number of virtual voices used is displayed in the corner of the voice graph.
RSP	this graph profiles the percentage of a frame period being used to execute the audio synthesis microcode on the RSP.

1157389

Table 20-9ie Profiled Resources

Profiled Resource	**Description**
CPU	this graph profiles the percentage of a frame period being used by the CPU during the call to alAudioFrame.
output meters	this profiles the peak output levels of the final output samples that are sent to the audio DACs. The scale is in dBs with the top of the meter at 0 dB and then decreasing in 3 dB increments per LED. Signal levels above -3 dB are indicated by a yellow caution LED. Signal presence is indicated by the bottom LED (ie any non-zero sample will turn on the bottom LED). Signal clipping is indicated by a red LED that appears above the meter. Note that the clip detector does not detect true clipping, rather it detects when a sample magnitude value of 0x7fff appears. This could be a legitamite value from a normalized sound or it could be a limited value caused by overflow.

Be aware that the resource demands for audio synthesis varies on a frame by frame basis. This is because it must share the processing resources with the other parts of the system. This means that the profile values will vary each time a given sequence is played. Therefore, the readings should be used as an approximation, not as an accurate measurement of resource usage. Also note that the CPU measurements can be affected by any debugging messages produced by the audio library. Also the N64 code was not optimized by gcord and so is not displaying best case performance.

The Nintendo 64 Menu

If the N64 development board is available, an N64 menu will appear in the editor. This menu provides control over some of the N64 functionality. The "Clear Profile Values" item resets the MIDI player and causes all the maximum values to be reset to zero. The "Configure Hardware" menu brings up a dialog which can be used to set some of the Audio Library configuration parameters. See Table 20-10 on page 428 for a description of the various configuration parameters. After setting the configuration parameters, press the okay or apply button to make the changes take affect. Reconfiguration may take a few seconds since any open bank file must be fully reloaded to the N64. Configurations can be saved and reloaded at a later time using the "Save Configuration..." and "Load Configuration..." commands. These commands ask you to name the configuration file you

115389

want to save or load before proceeding. Finally, the "Reset Hardware" command resets the entire N64 hardware forcing the N64 code to be reloaded and the audio reconfigured. Use this command to try to recover the N64 if it crashes for any reason.

Here is a description of each of the configuration parameters:

Table 20-10ie Configuration Parameters

Configuration Parameter	Description
output rate	the requested sampling rate of the audio interface in Hz.
samples per frame	the requested number of samples to be synthesized per audio frame. For maximum efficiency use a value that is a multiple of 160 samples (eg. 640). A larger number means a slower frame rate while a smaller number means a faster frame rate. This number, along with the output rate can be used to simulate a game running at 60 Hz or 30 Hz. For example, at an output rate of 44100 Hz, setting this value to be 735 will produce an frame rate of 60 Hz.
max commands per frame	the maximum number of ABI commands that can be executed per audio frame. This directly corresponds to the size of the audio command list buffer that stores the ABI commands.
DMA buffers	the number of available buffers for performing DMA requests.
DMA buffer size	the size of each DMA buffer. Smaller buffer sizes normally require more DMA requests while larger buffer sizes normally require fewer DMA requests.
max DMA requests	the maximum number of DMA requests that can be made. This value directly affects the size of the DMA message queue set up by the N64 code.
# frames to hold DMA buffers	the number of frames that must elapse before the N64 code will free a DMA buffer for reuse. While the buffer is being "held", its samples remain available for other requests that may ask for the same samples. In some cases, the same samples may be used over and over again so holding them in memory is faster than performing a DMA from ROM.
max virtual voices	the maximum number of virtual voices available to both the synthesis driver and the MIDI player.

Table 20-10 ie Configuration Parameters

Configuration Parameter	Description
max physical voices	the maximum number of physical voices available. If this is less than virtual voices then voice stealing is enabled.
max control updates	the maximum number of control updates each physical voice is able to store. Control updates store data such as volume changes, pitch changes, etc. This value directly affects the memory allocated for control data.
max channels	the maximum number of channels available for MIDI messages. Normal MIDI systems support 16 channels. This affects how much memory is allocated to store channel information.
max events	the maximum number of event updates that the synthesizer is able to store. Event updates store sequence data such as start commands, MIDI commands, etc. This value directly affects the memory allocated for event data.

Note that since audio sample DMA is implemented by the game application, the DMA configuration parameters may not be applicable to your game. Keep this in mind when setting these parameters.

Bugs

For a list of known bugs and problems, consult the man page for the ie tool.

1157689

Midi and the Indy

Before using Midi Daemon, you will have to correctly configure your Indy for midi. Because there have been changes in both the midi software, and the serial ports, on the motherboard, it is recommended that only a recently purchased Indy and the latest software releases be used.

Motherboards need to be of version 013 or newer. To determine the version of your motherboard, open your Indy, and on the front right of the motherboard, you will find a version number. The first four digits should be 8123 and they are followed by three more digits that are the version number. The revision number that follows the version number is not important. If you find that you have an Indy with an older version motherboard, contact SGI field service for a replacement board.

The Indy uses a standard Macintosh Computer Midi Interface. Because there are differences between the interfaces sold for the Mac, (particularly in the voltage levels necessary) not all Mac Midi Interfaces will work correctly. Insufficient testing has been done to recommend a particular brand. We have seen cases where interfaces that do not supply their own power, but instead draw their power from the Indy serial port will drop midi messages sent back to back. For that reason we do recommend that you purchase a midi interface that has its own power supply.

At present, we are recommending the installation of the DMedia 5.5 package, which contains the necessary midi drivers.

To configure your Indy for midi, you can use either of two methods. The first method, is to run startmidi. This utility is started from the command line, with arguments specifying which midi ports to turn on. This is the only way to turn on the internal midi port.

Alternately, you can turn on midi by using the Serial Port manager, in the System Manager tools. This provides a more user friendly interface, and once configured, a serial port will remain configured even after a reboot. If you find that selecting the System Manager or the Serial Port manager generates error messages pertaining to the object server, try the following sequence of commands:

```
/etc/init.d/cadmin stop

/etc/init.d/cadmin clean
```

11573769

```
/etc/init.d/cadmin start
```

You can verify that your midi is working, by starting Midi Daemon with the -v (verbose) option. If midi is working, you will get a message printed in the window for every midi message received.

If you wish to use serial port number one for receiving midi, it is important to turn off automatic spawning of getty's on that port. To do this, you must edit the file /etc/inittab. Find the line that starts with:

```
t1:23:respawn:/sbin/getty ttyd1
```

Change this to:

```
t1:23:off:/sbin/getty ttyd1
```

Save the file and reboot the Indy.

115731809

The sbc Tool

sbc

sbc is used to combine any number of MIDI sequences into a MIDI sequence bank (a .sbk file). A sequence bank file contains the sequences, one after the other (8-byte aligned), with a header at the front that allows indexing into the bank to retrieve individual sequences.

sbc is invoked as follows:

```
sbc -o <output file> file0 [file1 file2 file3 ....]
```

11573189

Chapter 21

Audio File Formats

1157189

This chapter describes the file formats used for Nintendo 64 audio development.

The first section details the bank format used by the Sequence Player. The second section provides information about the Standard MIDI File format as it relates to Project Reality.

Note: All multi-byte data types (short, long, and so on) are stored with the high byte first. This is the opposite of the Intel ordering found in PCs.

Bank Files

Bank files store the audio and control information needed to create audio from sequencer MIDI events. On the Nintendo 64, this information is encapsulated in two files: the bank file and the wavetable file.

The Bank (.bnk) file contains control information such as program number to instrument assignment, key mapping, tuning, and envelope descriptions. It is loaded into the Nintendo 64 DRAM during playback.

The Wavetable (.tbl) file contains ADPCM compressed audio data. Because of the size of the data, it is streamed into DRAM (and then to the RCP) only when it is needed.

The formats for both files are optimized for the Nintendo 64 to be efficiently used with the Sequence Player and the Sound Player. They are not intended to be interchange file formats, and contain no textual information or other data not directly related to playing back audio. Many features commonly found in standard patch and wavetable formats (for example, AIFF files) were sacrificed in favor of smaller files in ROM.

Note: References to objects are stored as offsets in the Bank files, but the alBnkfNew() call converts the offsets to pointers.

ALBankFile

Bank files must begin with an ALBankFile structure. This structure allows the software to locate data for a specific bank.

```
typedef struct {
s16revision;
s16bankCount;
s32bankArray[1];
} ALBankFile;
```

The ALBankFile fields are summarized in Table 15-1.

Table 21-1ALBankFile Structure

Field	Description
revision	File format revision number.
bankCount	Number of banks contained in the Bank file.
bankArray	Array of offsets of the ALBank structures in the bank file.

ALBank

The ALBank structure specifies the instruments that make up the bank, as well as the default sample rate and percussion instrument. Banks may contain any number of programs.

Note: The percussion field specifies an instrument for the Sequence Player to use as a default MIDI channel 10 (drum channel) instrument.

```
typedef struct {
s16instCount;
u8flags;
u8pad;
s32sampleRate;
s32percussion;
s32instArray[1];
} ALBank;
```

Table 21-2ALBank Structure

Field	Description
instCount	Number of programs (instruments) in the bank.
flags	=0 if instArray contains offset, and =1 if instArray contains pointers.
pad	Currently unused byte.

Table 21-2ALBank Structure

Field	Description
sampleRate	The sample rate at which this bank is inteded to be played.
percussion	The offset (or pointer) to the default percussion instrument.
instArray	Array of offsets (or pointers) to ALInstrument structures that make up this bank.

ALInstrument

The ALInstrument structure contains performance information.

```
typedef struct {
u8volume;
u8pan;
u8priority;
u8flags;
u8tremType;
u8tremRate;
u8tremDepth;
u8tremDelay;
u8vibType;
u8vibRate;
u8vibDepth;
u8vibDelay;
s16bendRange;
s16soundCount;
s32soundArray[1];
} ALInstrument;
```

Table 21-3ALInstrument Structure

Field	Description
volume	Overall instrument playback volume. 0x0 = off, 0x7f = full scale
pan	Pan position. 0 = left, 64 = center, 127 = right.

11513169

Table 21-3ALInstrument Structure

Field	Description
priority	The priority for voices for this instrument. 0 = lowest priority, 10 = highest priority.
flags	If soundArray values are offsets, flags = 0. If they are pointers, flags = 1.
bandRange	Pitch bend range in cents.
soundCount	Number of sounds in the soundArray array.
soundArray	Offsets of (or pointers to) the ALSound objects in the instrument.

ALSound

The ALSound structure contains information about the individual sounds that make up an instrument.

```
typedef struct Sound_s {
s32envelope;
s32keyMap;
s32wavetable;
u8samplePan;
u8sampleVolume;
u8flags
} ALSound;
```

Table 21-4ALSound STructure

Field	Description
envelope	Offset of (or pointer to) the ALEnvelope object assigned to the sound.
keyMap	Offset of (or pointer to) the ALKeyMap object assigned to this sound.
wavetable	Offset of (or pointer to) ALWavetable objects assigned to the sound.

11513189

Table 21-4ALSound STructure

Field	Description
samplePan	Pan position of the sound in the stereo field. 0 = full left, 0x7f = full right
sampleVolume	Overall sample volume. 0 = off, 0x7f = full scale.
flags	If envelope, keyMap, and wavetable are specified as offsets, flags = 0. If they are pointers, flags = 1.

ALEnvelope

The ALEnvelope structure describes the attack-decay-sustain-release (ADSR) envelope for a sound.

Note: Release volume is assumed to be 0.

```
typedef struct {
s32 attackTime;
s32 decayTime;
s32 releaseTime;
s16 attackVolume;
s16 decayVolume;
} ALEnvelope;
```

Table 21-5ALEnvelope Structure

Field	Description
attackTime	Time, in microseconds, to ramp from zero gain to attackVolume.
attackVolume	Target volume for the attack segment of the envelope.
decayTime	Time, in microseconds, to ramp from the attackVolume to the decayVolume.

Table 21-5ALEnvelope Structure

Field	Description
decayVolume	Target volume for the decay segment of the envelope. The sustain loop holds at the decayVolume.
releaseTime	Time, in microseconds, to ramp to zero volume.

ALKeyMap

The ALKeyMap describes how the sound is mapped to the keyboard. It allows the sequencer to determine at what pitch to play a sound, given its MIDI key number and note on velocity.

Note: C4 is considered to be middle C (MIDI note number 60).

Note: Bank files may not contain keymaps that have overlapping key or velocity ranges.

```
typedef struct {
u8 velocityMin;
u8 velocityMax;
u8 keyMin;
u8 keyMax;
u8 keyBase;
u8 detune;
} ALKeyMap;
```

Table 21-6ALKeyMap Structure

Field	Description
velocityMin	Minimum note on velocity for this map. 0 = off, 0x7f = full scale.
velocityMax	maxumum note on velocity for this map. 0 = off, 0x7f = full scale.
keyMin	Lowest note in this key map. Notes are defines as in the MIDI specification.

Table 21-6ALKeyMap Structure

Field	Description
keyMax	Highest note in this key map. Notes are defined as in the MIDI specification.
keyBase	The MIDI note equivalent to the sound played at unity pitch.
detune	Amount, in cents, to fine-tune this sample. Range is -50 to +50.

ALWavetable

The ALWavetable structure describes the sample data to be played for the given sound. It is described in detail below, along with the structures it contains.

```
enum     {AL_ADPCM_WAVE = 0,
          AL_RAW16_WAVE};

typedef struct {
    s32 order;
    s32 npredictors;
    s16 book[1];         /* Must be 8-byte aligned */
} ALADPCMBook;

typedef struct {
    u32         start;
    u32         end;
    u32         count;
    ADPCM_STATE state;
} ALADPCMloop;

typedef struct {
    u32         start;
    u32         end;
    u32         count;
} ALRawLoop;

typedef struct {
    ALADPCMloop *loop;
    ALADPCMBook *book;
```

1151319

```
} ALADPCMWaveInfo;

typedef struct {
    ALRawLoop *loop;
} ALRAWWaveInfo;

typedef struct {
s32base;
s32len;*/
u8type;
u8flags;
union {
ALADPCMWaveInfo adpcmWave;
ALRAWWaveInfo   rawWave;
} waveInfo;
} ALWaveTable;
```

Table 21-7ALWavetable Structure

Field	**Description**
base	Offset of (or pointer to) the start of the raw or ADPCM compressed wavetable in the table (.tbl) file.
len	Length, in bytes, of the wavetable.
type	the type (AL_ADPCM_WAVE or AL_RAW16_WAVE) of the wavetable structure.
flags	If the base field contains an offset, flags =0. If it contains a pointer, flags = 1.
waveInfo	Wavetable type specific information.

Table 21-8ALADPCMWaveInfo structure

Field	**Description**
loop	Offset or pointer to the ADPCM-specific loop structure.
book	Offset or pointer to the ADPCM-specific code book.

11573189

Table 21-9 ALRawWaveInfo structure

Field	Description
loop	Offset or pointer to the raw sound loop structure.

Table 21-10 ALADPCMLoop structure

Field	Description
start	Sample offset of the loop start point.
end	Sample offset of the loop end point
count	Number of times the wavetable is to loop. A value of -1 means loop forever.
state	ADPCM decoder state information.

Table 21-11 ALADPCMBook structure

Field	Description
order	Order of the ADPCM predictor.
npredictors	Number of ADPCM predictors.
book	Array of code book data.

Table 21-12 ALRawLoop structure

Field	Description
start	Sample offset of loop start point.
end	Sample offset of loop end point.
count	Number of times the wavetable is to loop. A value of -1 means loop forever.

11513189

ADPCM AIFC Format

The compressed ADPCM file format is based around AIFC. It uses a non-standard compression type and two application-specific chunks that contain the codebook and loop point information. This file is generated by the ADPCM encoding tool from standard AIFC and AIFF sample files, and is used by the Instrument Compiler to generate Bank and Table files.

As in AIFC, chunks are grouped together in a FORM container chunk:

```
typedef struct {
ID ckID; /* 'FORM' */
s32 ckDataSize;
s32 formType; /* 'AIFC' */
Chunk chunks[]
}
```

where `ckID` is always FORM and `formType` is AIFC. The standard AIFC chunks, which are essential, are the Common chunk, which contains information about the sound length; and the Sound data chunk.

```
typdef struct {
u32 ckID; /* 'COMM' */
s32 ckDataSize;
s16 numChannels;
u32 numSampleFrames;
s16 sampleSize;
extended sampleRate;
u32 compressionType; /* 'VAPC' */
pstring compressionName; /* 'VADPCM ~4:1' */
}
```

The current format accepts only a single channel. The `numSampleFrames` field should be set to the number of samples represented by the compressed data, not the the number of bytes used. The `sampleRate` is an 80 bit floating point number (see AIFC spec).

The Sound data chunk contains the compressed data:

```
typedef struct {
u32 ckID; /* 'SSND' */
s32 ckDataSize;
u32 offset;
```

11513189

```
u32 blockSize
u8 soundData[];
}
```

Both `offset` and `blockSize` are set to zero.

The encoded file will include two application-specific chunks. The common Application Specific data chunk format in AIFC is:

```
typedef struct {
u32 ckID; /* 'APPL' */
s32 ckDataSize;
u32 applicationSignature; /* 'stoc' */
u8 data[];
}
```

where `data[]` contains the application-specific data.

The Codebook application-specific data defines a set of predictors that are used in the decoding of the compressed ADPCM data.

```
typedef struct {
u16 version; /* Should be 01 */
s16 order;
u16 nEntries; /* 'stoc' */
s16 tableData[];
}
```

The `order` and `nEntries` fields together determine the length of the `tableData` field. In the current implementation, `order`, which defines the ADPCM predictor order, must be 2. `nEntries` can be anything from 1 to 8. The length of the `tableData` field is `order*nEntries*16` bytes.

The Loop application-specific data contains information necessary to allow the ADPCM decompressor to loop a sound. It has the following structure:

```
typedef struct {
u16 version; /* Should be 01 */
s16 nLoops;
```

11573189

```
adpcmLoop loopData[];
}
```

`nLoops` defines the number of loop points and hence the number of `adpcmLoop` structures in the chunk. In the current library, only one loop point can be specified. `loopData` has the following structure:

```
typedef struct {
u16 state[16];
s32 start;
s32 end;
s32 count;
} adpcmLoop
```

`state` defines the internal state of the ADPCM decoder at the start of the loop and is necessary for smooth playback across the loop point. The `start` and `end` values are represented in samples. `count` defines the number of times the loop is played before the sound completes. Setting `count` to -1 indicates that the loop should play indefinitely.

11573189

Sequence Banks

To provide a convenient way of collecting multiple MIDI sequences and accessing them from the ROM, Silicon Graphics has defined a simple Sequence Bank format. Files of this format are produced by the Sequence Bank Compiler (*sbc*), which takes multiple MIDI files and collects them with a simple header.

The format for the Sequence Bank file header is:

```
typedef struct {
u16 version; /* Should be 01 */
s16 seqCount;
ALSeqData seqArray[];
}
```

where `seqCount` is the number of sequences in the file, and the `seqArray` gives a list of offsets into the file and lengths for the individual sequences.

```
typedef struct {
u8 *offset;
s32 seqLen;
} ALSeqData
```

The offsets represent the position of the start of the sequence from the beginning of the file. Note that the start of all sequences are 8-byte aligned when the Sequence Bank Compiler is used.

1153189

Compressed Midi File Format

The compressed midi file format is composed of a header and up to sixteen individual tracks. Each midi channel will have its own track. If there are no midi events for a particular channel, the track will not be created, and the offset to that track will be set to zero.

The compressed midi file header is a collection of 16 offsets and a division value.

```
typedef struct {
    u32        trackOffset[16];
    u32        division;
} ALCMidiHdr;
```

The offset is specified in bytes from the begining of the file to the begining of the track. The division value is taken from the input midi file.

The format for the individual tracks is similar to the format used in a standard midi file. Each track consists of a series of events, seperated by delta times in ticks. Ticks are specified using variable length numbers, and ever event must have a delta value, even if that value is zero. Midi events are of the same format as that used in the standard midi file except as specified below.

1. There are no note offs, instead note ons are followed by a variable length number that specifies the number of ticks duration. As an example, a note on of middle C with a velocity of 80 and a duration of 240 ticks would be expressed by the following sequence of hex bytes: 0x90 0x3C 0x50 0x81 0x70. Note that when calculating the deltas between events, the duration is not taken into account.
2. Only two types of meta events are supported, tempo events and end of track events, and they are both slightly altered. Tempo events are composed of a meta status byte, (0xFF) a subtype byte (0x51) and three bytes that contain the new tempo. (Note that the len byte has been removed.) The end of track event is composed of only two bytes, a meta status byte, (0xFF) and a subtype byte (0x2F). Care should be taken to see that the end of track event occurs after all the notes in the track have played out their full duration.

1153189

3. Loops are allowed using a combination of loop start and loop end events. A track can have up to 128 loops which can be nested. Each loop within a track has a unique loop number. The loop start event is composed of four bytes; a meta status byte (0xFF), a loop start subtype byte (0x2E), a loop number (0-127), and an end byte (0xFF). A loop end event is composed of eight bytes, a meta status byte (0xFF), a loop end subtype byte (0x2D), a loop count byte (0-255), a current loop count (should be the same as the loop count byte), and four bytes that specify the number of bytes difference between the end of the loop end event, and the begining of the loop start event. (note that if this value is calculated before the pattern matching compression takes place, this value will have to be adjusted to compensate for any compression of data that takes place between the loop end and the loop start.) The loop count value should be a zero to loop forever, otherwise it should be set to one less than the number of times the section should repeat. (i.e. to hear a section eight times, you would set the loop count to seven.)
4. Running status is supported for all events except across meta events and across loop points.

The compressed midi file format uses a system of matching patterns in the data, and replacing them with markers, instead of repeating the data. When constructing tracks, any pattern of data may be replaced by any previous track data with a marker. A pattern marker consists of four bytes. The first byte is 0xFE. The second two bytes are an unsigned 16 bit value that specifies the difference, in bytes, between the begining of the marker, and the begining of the pattern. The last byte is the length of the pattern. In order to distinguish between a data byte of 0xFE and a pattern marker's first byte, any data byte of 0xFE will be followed by another byte of 0xFE.

Note: The maximum pattern length is 0xFF and the maximum distance between the marker and the pattern is 0xFDFF.

Nesting of patterns is not supported. If a marker is encountered within a repeated pattern, the marker data will be returned to the sequence player, as actual midi data.

Note: Patterns replaced with markers may not contain bytes of value 0xFF or the current loop count byte of a loop end event.

1151359

Chapter 22

Nintendo 64 Audio Memory Usage

The following sections discuss the memory used by the audio system in a typical application. Memory requirements, and optimization are discussed in detail.

11573189

Overview of audio RDRAM usage.

The amount of RDRAM needed by the audio system is dependent on numerous factors. Most importantly, the number of sounds being played at any given time will determine the size of most buffers. Most buffers must be large enough to accommodate the worst case scenario. Applications with fewer voices will need fewer buffers. The sample rate and frame rate chosen will effect the size of several important buffers.

Audio Buffers

The majority of memory used by the audio, that can be optimized, comes from the following buffers:

- The Sample DMA Buffers.
- The Command List Buffers.
- The Audio Output Buffers.

There are several other buffers, but the gains obtained by optimizing them are less significant. These include:

- The Audio Thread Stacksize.
- The Synthesizer Update Buffers
- The Sequencer Event Buffers

In addition to optimizing the buffers listed above, it is important that several other buffers are no larger than they need to be. While you can't optimize them per se, you should check to make sure that their size is no bigger than need be. Important buffers of these type include:

- The Audio Heap,
- The Sequence Buffer
- The Bank Control File Buffer
- The Reverb Delay Line Buffer

Because the heap size is dependent on the size of the buffers allocated from the heap, it is important to optimize the other buffers first.

1153709

Sample Rate, Frame Rate, and Other Factors

In order to determine the size of most of the buffers, you will need to determine several factors first. Most importantly, sample rate and frame rate. Higher sample rates will require larger output buffers, more DMA space, and larger command list buffers. Likewise, slower frame rates require larger output buffers, more DMA buffer space, and larger command list buffers.

Note: Audio frame rate can be different from video frame rate. It is possible for the audio to be operating at 60 frames per second, while the graphics are operating at 30 frames per second.

In addition to the sample rate and frame rate, the specific sounds, and how they are set up can effect the size and number of DMA buffers, as can the individual sequences used.

1157399

Optimizing Buffer Sizes.

Audio DMA Buffers

The first area to try and optimize is the number of DMA buffers. These buffers are used by the audio synthesizer to store samples from the cartridge during creation of the output buffers. In the worst case scenario you will need four buffers for every voice you have allocated. However, in practice you need only a portion of that. The actual number of buffers you will need is very dependent on the sequences and sound effects played. To optimize this value, you will need to allocate sufficient buffers to keep from crashing, and then play your game for a while. At the end of each frame you should be calling a routine that frees DMA buffers that have become stale. (Called `__clearAudioDMA` in example programs.) In this routine, before discarding stale buffers, step through the list of used DMA buffers and count how many there are. If you keep track of the maximum value, you can report this at the end of game play, using your choice of debugging method. The following code is an example of how to perform this count.

```
#ifdef AUD_MEM_PROF
    ampDMAcount = 0;
    dmaPtr = dmaState.firstUsed;
    while(dmaPtr)
    {
        ampDMAcount++;
        dmaPtr = (AMDMABuffer*)dmaPtr->node.next;
    }
    if(ampDMAcount > ampMaxDMABufs)
        ampMaxDMABufs = ampDMAcount;
#endif
```

Because the number of buffers used can vary slightly, even when playing the same music and sound effects, it is always a good idea to have a few more buffers than you ever found yourself needing.

In addition to the number of DMA buffers needed, it is helpful to know what is the maximum number of DMA's performed in any frame. This number will allow you to optimize the number of DMA message buffers you will need. Because the size of a message buffer is substantially less than the size of a DMA buffer, the result of this optimization is not much. However, it is easily performed since there is a variable that reports the number of DMA's done each frame. All you need to do is record its maximum value, checking it once a frame, and then report that value at the same time you report the number of DMA buffers used.

Another place for optimization is the length of the DMA buffers. Longer buffers will require fewer buffers, and use fewer DMA's. Conversely, smaller buffers will require more buffers and more DMA's. Generally, the smaller buffers, even though more are required, will use memory more efficiently. However, the smaller buffer sizes will also generate more DMA's and for that reason are less efficient in terms of processing time. It is up to the developer to decide what trade off between memory usage and processing time to pick. Optimal buffer sizes are probably ones that will handle enough samples to process one frame of audio. Below, is a table that compares the same music played back with various buffer sizes. (All other factors were the same.)

Table 22-1 DMA Buffer Length.

DMABufLength	MaxDMA/Frame	MaxDMABuffers	BufLen*MaxBufs
0x600	12	26	39936
0x500	12	30	38400
0x400	14	34	34816
0x300	16	38	29184
0x280	17	43	27520
0x200	22	50	25600

As can easily be seen, the amount of buffer space needed goes up as the size of the buffers go up, even though fewer buffers are needed. However, at the same time, the number of DMA's goes down. In this case, probably the value of 0x500 is optimal, since it causes the least number of DMA's per frame in the worse case situation, but allows the memory allocated to buffers to be smaller than it would be with buffers of 0x600 size.

Another constant that can be changed is `FRAME_LAG`. This value defines how long a DMA buffer will be kept after it has been used. If you continually use the same sample, that sample will be kept in memory, and will not need

1157369

to be DMA'ed again. Higher lag values will lower the number of DMA's but will increase the number of DMA buffers needed.

Command List Size

Like the number of DMA buffers, the command list size is dependent on the sequences and sound effects used by the game. To optimize the command list size, simply record the maximum value used, and check that value at the end of game play. Because this can vary, even when playing the same audio, it is wise to leave a little more than you ever needed.

Output Buffer Size

The output buffer size is determined by the audio playback rate, and the frame rate. If you synch audio to the vertical retrace you will need to have three audio output buffers. If you synch the audio to the audio completion interrupt, you will only need to have two output buffers. Example code is included in the example applications demonstrating calculating the size of the output buffers.

Audio Thread Stacksize

The audio thread stacksize can be determined using the `stacktool`, and optimized accordingly.

Synthesizer Update Buffers and Sequencer Event Buffers

Synthesizer update buffers and sequencer event buffers are allocated from the audio heap when the synthesizer and sequencer are created. There is, at present, no way to efficiently optimize these values. However, because the size of each buffer is small, it is better to allocate a few too many, than not enough.

11573189

The Audio Heap

Once all calls to `alHeapAlloc` have been completed, you can determine the amount of the heap that has been used by subtracting the heap's current value from the heap's base value. These values are part of the heap structure.

The Sequence Buffer

The sequence buffer needs to be large enough to hold the largest sequence that will be used.

The Bank Control File Buffer

The bank control file buffer needs to be large enough to hold the bank control file. This is the <bank>.ctl file.

115731389

11573189

Chapter 23

Using The Audio Tools

115731889

This chapter instructs the musician and sound designer in how to use the audio development tools currently available for the Nintendo 64. It is divided into the following sections:

- An overview of the audio system.
- Discusion of the constraints and decisions that should be made in conjunction with the programmer or game designer.
- Suggestions for creating samples.
- Playback parameters and the .inst file.
- How to create bank files.
- MIDI files and MIDI implementation.
- Music development tools.

Overview of Audio System

In order for the musician or sound designer to produce sounds and music for the Nintendo64, a short explanation of the audio system is helpful, though not necessary. To that end, a brief description of the audio system is included here. (The audio system is discussed in greater detail in the programmers documentation.) In addition to a brief description of the audio system, several important items the musician should be aware of are listed below.

Brief description of audio system

The audio system for the Nintendo 64 is composed of a Sound Player (for playing single samples, such as sound effects) and a Sequence Player (for playing music). When the game starts up, it creates and initializes a sound player and a sequence player. It then assigns a bank of sound effects to the sound player, and assigns a bank of instruments and a bank of MIDI sequences to the sequence player. To play a sound effect, the game sends a message to the sound player, telling it what sound effect to set as its target, and then sends another message to the sound player, telling it to play the target sound. To play a MIDI sequence, the game must load the sequence data, then attach the sequence to the sequence player, and then send a message to the sequence player to start playing the music.

Note: Musical sequences can be stored as either type 0 MIDI files, or in a compressed midi format unique to the Nintendo64. It is very important that the programmer and the musician agree on which file format to use.

There are several components to the sound system. First, there are the samples that are stored in ROM. Accompanying the samples are a group of parameters used for playback (Key Mappings, Envelopes, Root Pitch, and so on). In order to process the sounds, a section of the RAM must be allocated for the audio system.

In software, there are two main sections. One part runs on the CPU and the other part runs on the RSP. The audio system must share the RSP with the graphics processing. The RSP is where most of the low-level processing takes place, and this is where the samples are mixed into an output stream. This output stream is then fed to a pair of DACs for stereo output.

There are four types of files used by the game for audio production: .ctl, .tbl, .seq, and .sbk. Before the game can play back either sound effects or music, the musician and sound designer must create these files. The .tbl files contain the compressed samples. The .bnk files contain the associated control information necessary for playback. .bnk files and .tbl files are always paired.

The .seq files are MIDI files that have all unneeded events removed, and the .sbk files are banks of .seq files. Typically, there will be at least one pair of .bnk and .tbl files for music, and a seperate pair for sound effects. (Although it would be possible to put all sounds into one pair, or alternatively, have numerous pairs.)

The reason that banks are stored in two files is that then the raw audio data doesn't need to be loaded into RAM; only the information pointing to the samples, and the values for the playback parameters. When a sound is to be played, only a small portion of the sample is loaded into a RAM buffer. After it has been used for playback, it can be discarded, and the buffer reused for the next portion of the sample. The result is that a comparatively small amount of RAM is needed for sound.

Typical Development Process

When creating audio for an Nintendo 64 game, the musician typically follows these steps:

1. Create the samples as AIFF files.
2. Encode the samples into AIFC files.
3. Create a .inst file.
4. Compile the .inst file, with the samples into the bank files.
5. Create the MIDI sequence files.
6. Compile the MIDI sequence files into .seq files, and then compile the .seq files into a .sbk file.
7. Deliver the .tbl .bnk and .sbk files to the programmer.

1157189

Common Values

Throughout this document and when referring to .inst files, several things are kept constant:

- Middle C (MIDI note 60) is referred to as C4. (Some synthesizer and software manufactures refer to Middle C as C3.)
- Pan values range from 0 to 127, with 0 being full left, 64 center pan, and 127 full right.
- Volumes are from 0 to 127, with 0 meaning there will be no sound, and 127 being full volume.

115731899

11513789

Dealing With Constraints and Allocating Resources

When you use the Nintendo 64 system, there are several choices that you must make. Most of these choices center around how to use the fewest system resources, while still maintaining a sufficient level of quality. Unconstrained by limits on available resources, the Nintendo 64 system is capable of audio rivaling top-of-the-line samplers.

Most of the limits in the software system are easily changed. However, in most cases a great deal of time can be saved if the programmer, game designer, and musician all agree beforehand what these values are going to be set to.

The limits on resources will fall into several categories:

- determining hardware playback rate
- limits of voices and processing time
- division of sounds and music into banks
- limits of ROM space

Determining Hardware Playback Rate

The principle decision to make about software is deciding what playback rate the hardware should be set to. Typically, rates from 22050 Hz to 44100 Hz are chosen. Higher rates require the software to produce more samples, and consequently take more processing time. Although there are no hard rules to follow, values of 44100 Hz are ideal, but values of 32000 Hz and 22050 Hz do not produce a substantial loss of audio quality. Values below 22050 Hz quickly begin to degrade the quality of the audio.

Also of considerable importance is the fact that samples sound better if the output rate is as close as possible to their sample rate. If all the samples in the game are sampled at 22050 Hz, the output quality will be best with a playback rate of 22050 Hz. If there is uncertainty in the planning process, it is better to start with a higher rate, and resample down later, than to start with a lower rate and resample up later.

Limits of Voices and Processing Time

The factor limiting the number of voices available for playback is the amount of time the audio will have for processing. Obviously, the more voices, the more processing time needed, and the higher the audio playback rate, the more time needed. As a rough guideline, it is estimated that 1% of RSP time is needed for each voice, when playing at 44.1k. So, if the audio is given 20% of RSP processing time, then fifteen to twenty voices will be possible. However, if the audio is given 40% of processing time, then 30 to 40 voices will be possible. Remember that a lower output playback rate reduces processing time, thus increasing the number of voices available for playback.

Division of Sounds and Music Into Banks

There are no formal rules specifying how the sounds and music will be organized. However, in most cases it is best to organize the sound effect samples into a bank (or banks) separate from the music samples.

There are two ways that the sequences may be stored in the game. They may be stored as separate sequences, or they may be compiled into a .sbk file. The music samples and MIDI files should be organized so that each sequence (or, if used, each bank of MIDI files) has a corresponding bank of music samples. If samples are shared by different MIDI files, they should be stored in the same bank. If the sequences do not share the same sample bank, duplicates of the samples will be produced in the different bank files.

Limits of ROM

The amount of space available for audio is strictly up to the game developer.

1151369

Creating Samples

Creating samples for the Nintendo 64 is similar to creating samples for any sample player. However, there are several additional facts to keep in mind.

To be recognized by the ADPCM tools, the samples should be stored as AIFF files, or uncompressed AIFC.

Samples benefit from being sampled at the same sample rate as the output playback sample rate. Because all samples are compressed with a variation of ADPCM, when they are played back at rates significantly different from their sampled rate, the noise can become rather obvious.

As an example, if the output sample rate is set to 44100 Hz, but the sample is sampled at only 22050 Hz, then to playback the sample at its original pitch, the sample converter must create two samples from each sample. Worse, if the sample is to be played an octave below its original pitch, the sample converter must create four samples for each sample. Because of the noise and distortion introduced from ADPCM, this will not be nearly as good quality as it would be if samples were recorded at 44100 Hz, or if the output playback rate were changed to 22050 Hz. For this reason, you may want to resample all samples to match the output sample rate, before performing the ADPCM conversion

Samples may be looped at any location in the sample. Although many ADPCM systems require you to loop samples at specific boundaries (the Super Nintendo, for example, required that loop points be multiples of 16), the Nintendo 64 makes no such requirement. If a sound is looped, it will loop as long as the sound is playing. When a looped sound's envelope enters the release phase, then the sound will still continue to loop.

All looped samples should last until the next multiple of 16, after the loop end. (This is because the ADPCM encoding stores the samples in blocks of 16.) For this reason, it is prudent to leave at least 16 samples after the loop end, on any sample that loops. As a nice feature, the adpcm tools provided have an option that truncates any sample to the shortest viable length, so there is no benefit to the musician calculating and truncating looped samples.

In other words, when creating looped samples, find your loop points, and don't worry about the release portion of the sample. If you want to truncate the sample, to keep samples on your hard disk smaller, but always leave at least 16 samples after the loop end. Then when you encode the samples, make sure you use the -t option, and the samples will be automatically truncated for you.

115731809

1151389

Playback Parameters and .inst Files

This section contains information about how to create the .inst file.

Setting Sample Parameters in the .inst File

In order for the Nintendo 64 audio system to playback samples correctly, it must have information for controlling aspects such as pitch and volume. These parameters are set by creating and editing a .inst file. Although some discussion of parameters follows, it is highly recommended that you review an example .inst file, because many of the parameters will be much clearer then.

The .inst file is a collection of objects, defined by text using C language syntax. The objects are:

- envelopes
- keymaps
- sounds
- instruments
- banks

The objects are related as follows: The basic unit representing a sample is a sound. That sound has an associated keymap, which specifies the velocity range, key range, and tuning of the sample. Also, the sound has an associated envelope that specifies the ADSR used to control the sample's volume. Sounds can be grouped into an instrument. Instruments are then grouped into a bank. Currently, there is only one bank in a .inst file. Because program control changes are limited to values from 1 to 128, MIDI sequences can only use the first 128 instruments in a bank. Game applications can select higher values by calls to the audio API.

Differences Between Sound Player and Sequence Player Use of .inst Files

The sound player and sequence player use the bank files created from the .inst files in different ways. While the sequence player uses the bank to identify instruments, and then uses the keymaps to identify which sound to play for what MIDI notes, the sound player does none of this. The sound player does not use the bank structure, the instrument structure, or the keymap parameters. However, for the .inst file to compile, every .inst file must have a bank and an instrument. Also, every sound must point to a keymap. This keymap may be shared by all the sounds in the .inst file, so only one keymap is needed.

For these reasons, the example .inst sound effects files are set up with one bank, with only one instrument, that lists the sounds in sequential order. There is no concern for overlapping of keymaps in this case, because the sound player ignores them. However, there is one default keymap, in order to allow the file to compile. In order for the pitch of a sound effect to be altered from its recorded pitch, the application must set the pitch, not the .inst file.

Envelopes

The Nintendo 64 audio system supports the use of ADSR envelopes for controlling volume. Envelope time values are in microseconds. (Because microseconds are a much finer control than most synthesizers and samplers use, musicians will have to adjust their thinking to accommodate much larger numbers than are usually used by samplers. Remember, an attackTime of 100,000 will produce an attack of one tenth of a second.) Maximum volume values are 127. In order to avoid any pops or clicks at the ends of sounds, you should always end an envelope with a release volume of zero. This is particularly true in the case of looped samples.

When using the sound player to play sound effects, if the decay time is set to -1, then the envelope will never enter the release phase. (In other words, it will loop forever.) To stop the sound, the game will have to call alSndpStop().

Keymaps and Velocity Zones

Note: Keymaps are used only by the sequence player. They are ignored by the sound player.

In addition to an envelope, every sample has a keymap. This keymap defines what keys and velocities the sample will respond to. By using

1153399

different keymap settings, it is possible to create instruments that play different samples for different keys and velocities.

In the keymap object, you set the minimum and maximum velocity values, as well as the minimum and maximum keys to respond. Note that you cannot create overlapping keymap zones. When the sequence player is trying to map a note to be played, it will search through the possible keymaps, and when it finds one that it can use, it will not continue to search.

Note: The Nintendo 64 imposes an upper limit on the keyMax value of one octave more than the keyBase.

Tuning for Samples Recorded at the Hardware Playback Rate

In addition to the velocity and key zone information contained in the keymap structure, all samples have a keyBase and a detune value. The keyBase sets the sample's pitch in semitones, and the detune value is used to fine-tune the sample in cents. (A cent is 1/100th of a semitone.) If the sample rate of the sound matches the hardware playback rate, the keyBase is the MIDI note value of the sample's original pitch. If the sample rate does not match the hardware playback rate, the keyBase must be altered to compensate for the difference in rates.

As an example, if a note of F4 is recorded at 44100, and the playback rate is also 44100, then the keybase should be set to 65 (since 65 is equivalent to MIDI note F4) and the detune is set to zero.

Tuning for Samples Recorded at Varying Rates

One of the more complicated aspects of the .inst files is the tuning of samples that are not sampled at the same rate as the hardware output rate. (remember that the hardware output rate is determined by software, and can be changed). Although the sample rate will be extracted from the AIFF file, you must adjust the keyBase parameter and the detune parameter if you want the sample to play back at the correct pitch.

In order to calculate keyBase and detune from a given sample rate, use the following formula:

N = semitones to add to keybase

N= 12log2(HardwareRate/SampleRate)

A much easier way to deal with the tuning issue is to use Table 16-1. In this case, pick an acceptable rate from the column that corresponds to your hardware rate. Record your sample at that rate (or resample your sample at that rate), and then add the number of semitones in the leftmost column to the MIDI note value of the samples pitch. Notice that this method insures a value of zero for the detune.

As an example, suppose that you had a hardware playback rate of 44100, but you wished to critically resample a sample of a trumpet playing Bb4 to a sample rate of about 32000 Hz. Instead of using 32000, you would resample to a rate of 33037, and then in your .inst file, you would add 5 semitones to the midivalue. Since Bb4 is the same as MIDI note number 70, you would add 5 and your keyBase value would be 75.

Table 23-1 Tuning to hardware playback rates.

Add to MIDI Value	Hardware Playback Rate of 44100	Hardware Playback Rate of 32000	Hardware Playback Rate of 22050
0 semitones	44100	32000	22050
1 semitones	41624.857	30203.978	20812.429
2 semitones	39288.633	28508.759	19644.317
3 semitones	37083.532	26908.685	18541.766
4 semitones	35002.193	25398.417	17501.097
5 semitones	33037.671	23972.913	16518.836
6 semitones	31183.409	22627.417	15591.705
7 semitones	29433.219	21357.438	14716.609
8 semitones	27781.259	20158.737	13890.626
9 semitones	26222.017	19027.314	13111.008
10 semitones	24750.288	17959.393	12375.144

1157389

Table 23-1 (continued) Tuning to hardware playback rates.

Add to MIDI Value	Hardware Playback Rate of 44100	Hardware Playback Rate of 32000	Hardware Playback Rate of 22050
11 semitones	23361.161	16951.410	11680.581
12 semitones	22050	16000	11025

To extend the above table, or produce a table with a different hardware playback rate, use the following formula:

Sample Rate = S

Hardware Rate = H

Number of semitones to add to MIDI value = N

$$S = \frac{H}{2^{N/12}}$$

Sounds

A sound structure is simply a reference to the sample, the keymap, the envelope, a value for pan, and a value for volume. Pan values are in the range of 0 to 127, with 0 equal to full left, 64 equal to center pan, and 127 equal to full right. Volumes are specified by values of 0 to 127.

Instruments

The instrument structure is a list of sounds grouped into an instrument. If the instrument is a musical instrument to be used by the sequence player, it is limited to 128 sounds, since that is the maximum number of MIDI notes. However, if the instrument is for use by the sound player, it may have as many sounds in it as you like. In addition to the list of sounds, the instrument has an overall volume and pan. (The sound player ignores these volume and pan values. Instead the sound player uses the pan and volume values specified in the sound object.)

1157389

The instrument structure can be used to create Drum Kits. In this case, you create an instrument that uses multiple sounds and associated keymaps. (There is a good example of this in the General MIDI Bank provided with the developer's package.)

Banks

At the top level of the .inst file is the bank structure. A .inst file may contain as many banks as needed. The bank must be selected by the application, since there is currently no way to switch banks via MIDI.

Creating Bank Files

The process for creating sample bank files is as follows:

1. Record the samples and save as .AIFF files.
2. Encode the samples using tabledesign and vadpcm_enc.
3. Create the .inst file.
4. Compile the bank using ic.

1157389

MIDI Files

Sequences can be stored in the game in one of two ways. Either as MIDI file Type 0, or in a compressed MIDI file format. To use MIDI Type 0, save the file as either a Type 0 or Type 1 MIDI file, and then use midicvt. To use the compressed sequence format, save the file as either a Type 0 or Type 1 MIDI file, and then use midicomp.

The process for creating MIDI sequence bank files is as follows:

1. Create the sequences and save them as MIDI files of either Type 0 or Type 1.
2. Convert the sequences using either midicvt or midicomp.
3. Compile the sequences using sbc.

The following MIDI messages are supported by both file formats:

- Note on
- Note off
- Polyphonic key pressure
- Midi Controllers:
    - Controller 7: Channel volume
    - Controller 10: Channel Pan
    - Controller 64: Sustain
    - Controller 91: FXMix
- Program Control changes 0-127
- Pitch Bend Change

In addition to the above MIDI messages, the MIDI file meta tempo event is supported.

Loops in the sequences.

The way loops are implemented in the two sequence formats are very different. If a game uses MIDI Type 0 format, the loops must be created by

1157389

the programmer using audio library calls from within the game code. If the compressed sequence type is used, loops are inserted by the musician. This is done using midi controllers.

The compressed sequence format supports looping within tracks. A track can have as many as 128 loops, which can be sequential or nested. Each loop is numbered, and must have a loop start and a loop end. Optionally, it can have a loop count, that specifies the number of times the looped section should play. Loop counts are limited from 1 to 255. A loop count of zero, the default, will loop forever.

Although the format used in the compressed midi file is not detailed here, it should be noted that when a file is compressed, midi events are rearranged into tracks based on channel. All midi events for channel 1 are put in the first track, and all midi events for channel 2 are put in the second track, and so on. This is particularly important when considering loops. If a loop is put in a track, all midi events from that channel will loop.

To insert loops into a compressed midi sequence, you will need to insert extra controllers. These controllers serve as markers for the loop. A loop start is defined as a controller number 102. A loop end is defined as a controller 103. Within a channel, each loop start and loop end pair must have a unique number between 0 and 127. This number is what the loop start and loop end controller's value should be set to. A loop count between 0 and 127 is created with a controller 104, using values 0 to 127. A loop count between 128 and 255 is created using controller 105, with values 0 to 127. (When a loop count controller 105 is encountered, the value is added to 128 to produce loop counts from 128 to 255.)

As a simple example, consider the following sequence:

```
loop 0 start        (controller 102 with value 0)
loop count of 6     (controller 104 with value 6)
loop 0 end          (controller 103 with value 0)
```

In this case the section between the loop start and the loop end will be played six times.

It is important to understand that the loop count is not associated with a start and end pair. When a loop end is encountered, it uses the most recent loop

1153169

count, even if there has already been a loop end for another loop. Consider the following sequence:

```
loop 0 start   (controller 102 with value 0)
loop count of 8(controller 104 with value 8)
loop 0 end     (controller 103 with value 0)
loop 1 start   (controller 102 with value 1)
loop 1 end     (controller 103 with value 1)
```

In this case, the first loop (loop 0) will have a loop count of 8. The second loop (loop 1) will also have a loop count of 8, since once set, the loop count continues until changed. If there has never been a loop count in the sequence, the loop count is set at its default of 0, which is interpretted as loop forever.

Warning: **All loops must have a loop start and a loop end with at least one valid midi event in between.**

Nesting Loops.

In the compact sequence format it is easy to nest loops. Consider the following sequence:

```
loop 0 start   (controller 102 with value 0)
loop 1 start   (controller 102 with value 1)
loop count of 8(controller 104 with value 8)
loop 1 end     (controller 103 with value 1)
loop 2 start   (controller 102 with value 2)
loop 2 end     (controller 103 with value 2)
loop 3 start   (controller 102 with value 3)
loop count of 4(controller 104 with value 4)
loop 3 end     (controller 103 with value 3)
loop forever   (controller 104 with value 0)
loop 0 end     (controller 103 with value 0)
```

In this case loop 1 will loop eight times, before the sequence proceeds to loop 2, which will also loop eight times. After that, loop 3 will loop 4 times, and then the entire sequence will loop infinitely.

1153189

Putting Things Together Into Makefiles

In the developer's kit, there is a directory named viper that shows how files would be arranged to build a bank of music samples. The makefile in this directory shows examples of setting up rules for files, and dependencies in a logical order. When you start a project, you can use these files as a template.

1157318

General MIDI and the Nintendo 64

Although the Nintendo 64 is not specifically a General MIDI device, it can be configured as one. As part of the developer's kit, there is a General MIDI Bank that demonstrates this. All the sound files used in this bank are also provided and may be used by licensed developers in any Nintendo 64 project.

Currently, MIDI channel 10 is configured to default to program 128. In the General Midi Bank, this is the Standard Drum Kit. If you send a program change on channel 10, the specified program will be selected, and channel 10 will no longer be the Standard Drum Kit.

11573189

11573189

Chapter 24

Scheduling Audio and Graphics

1153189

The Nintendo64 audio and graphics chores are shared between the host CPU and the RCP. The work to be performed is expressed using an array of primitives called a *command list*.

The host CPU is responsible for *command list generation*. Audio command lists are generated by calling alAudioFrame(). Graphics command lists are generated by calling the various graphics macros defined in gbi.h. In addition, the host CPU is responsible for assembling command lists into RCP *tasks* (which consist of command lists, RCP microcode and execution state information), and for downloading the task at the appropriate time to the RCP.

The RCP is responsible for *command list processing*. The RCP microcode loaded by the host CPU parses the command list, executes the appropriate core rendering routines, and writes the results to the video frame or audio buffer.

Since the video frame buffer must be updated at a regular rate (usually 30 frames per second) and the audio buffers must be updated before they are emptied by the audio DAC to prevent clicks and pops, the application must make schedule the command list generation and processing chores so that they happen in a "timely manner". This chapter identifies the relevant scheduling issues and describes the libultra *Scheduler* that addresses them.

1157319

Scheduling Issues

Command List Generation

Command lists are usually generated during the frame before they are to be processed. Though command list generation should take less than a frame time to complete, there are infrequent occasions when it may take longer. When the host CPU misses its completion deadline, *host overrun* is said to have occurred.

The effects of host overruns are usually undesirable. If an audio command list is not ready to be processed during the next frame time, clicks and pops will be introduced into the audio stream. If a graphics command list is not ready to be processed, the video frame buffer will not be updated until the following frame, which may cause the graphics stream to appear "jerky".

The effects of host overruns on the audio stream can be minimized if the audio and graphics command lists are generated in separate threads. Specifically, if the audio thread runs at a higher priority than the graphics thread, the host CPU can schedule the audio task even though the graphics task may not be completely generated, preventing clicks and pops from being introduced into the audio stream.

Alternately, one could implement a dynamic buffering scheme to prevent overrun by dynamically varying the audio data buffer size to accommodate any graphics overrun. This approach would require somewhat larger buffers and is more difficult to implement since overrun is dependent on things that are not known until runtime.

Note: Calls to alAudioFrame() generate DMA requests, which are assumed to be complete when the audio command list is processed. The DMA latency depends on the operation of the audio DMA callback which is implemented by the application.

Command List Processing

While audio command list processing time is deterministic (based on the number of active voices), the graphics command list processing time is variable (based on the complexity of the scene and the perspective of the viewer). Unless great care is taken in the construction of the graphics command lists, they may require more than a frame time to process. This is call *graphics (RCP) overrun*.

The effects of graphics overrun can be minimized by suspending the overrunning task and running the waiting audio task at the beginning of a video frame. Graphics tasks can be suspended with the osSpTaskYield() function. See the osSpTaskYield man pages for more information.

11573189

Using the Scheduler

The Scheduler is a host CPU thread that addresses the issues discussed above. It is responsible for executing audio and graphics tasks on the RCP such that host and RCP overrun is minimized or eliminated.

Each video retrace, the Scheduler reads the new tasks generated by client threads from the task queue and adds them to the end of a real-time (audio) or non-real-time (graphics) task schedule list.

If the previous frame's graphics task has overrun, the Scheduler causes the task to yield. It then runs the next audio task, resuming the yielded task when the audio task has completely processed, and any additional graphics tasks that are to be run to be run in the current frame.

When a task completes, the Scheduler sends a message to the client indicating that the work it requested is complete.

Creating the Scheduler: osCreateScheduler()

In order to use the Scheduler, you must first call osCreateScheduler() to initialize the OSSched data structure, its message queues and the Vi Manager. The osCreateScheduler() function spawns a thread to schedule and manage task execution. One of the parameters to this call is the thread priority, which should be higher than that of the threads which generate the command lists.

Adding Clients to the Scheduler: osScAddClient()

The Scheduler instantiates the Vi Manager and receives all retrace messages. However, clients of the Scheduler can receive a copy of the retrace message by providing a message queue when they sign in. This is accomplished by calling the osScAddClient() function.

Note: One of the parameters to this call is the message queue on which you wish to receive retrace messages. Make sure that the queue is big enough if you don't want to lose messages, as the Scheduler does not block when the queue is full.

1153189

Creating Scheduler Tasks: The OSScTask Structure

In order to send tasks to the Scheduler for execution, you must first create and initialize an OSScTask structure. The structure and a description of its fields is listed below.

```
typedef struct OSScTask_s {
    struct OSScTask_s    *next;
    s32                  state;
    u32flags;
    void*framebuffer;

    OSTask list;
    OSMesgQueue*msgQ;
    OSMesg msg;
} OSScTask;
```

Table 24-1OSScTask structure fields

Field	Description
next	Not used by client (used by the scheduler for list management).
state	Not used by client (used by the scheduler for state management).
framebuffer	Address of the frame buffer for this task (if it is a graphics task).
list	Structure containing task code and command list data (described below).
msgQ	The message queue on which the client is to receive the task done message.
msg	The message that the client is to receive when the task in done.

11571389

1157319

Table 24-2OSTask structure fields

Field	Description
type	Task type; should be initialized to M_AUDTASK for audio tasks or M_GFXTASK for graphics tasks.
flags	Various task state bits; should be initialized to 0 for audio tasks, or OS_TASK_DP_WAIT for most graphics tasks
ucode_boot	Pointer to boot microcode; should be initialized to rspbootTextStart.
ucode_boot_size	Pointer to boot microcode size in bytes; should be initialized to ((u32)rspbootTextEnd - (u32)rspbootTextStart).
ucode	Pointer to task microcode. Should be set to one of gspFast3DTextStart, gspFast3D_dramTextStart, gspLine3DTextStart, or gspLine3D_dramTextStart for graphics tasks; otherwise aspMainTextStart for audio tasks.
ucode_size	Size of microcode; should be initialized to SP_UCODE_SIZE.
ucode_data	Pointer to task microcode. Should be set to one of gspFast3DDataStart, gspFast3D_dramDataStart, gspLine3DDataStart, or gspLine3D_dramDataStart for graphics tasks; otherwise aspMainDataStart for audio tasks.
ucode_data_size	Size of microcode data; should be initialized to SP_UCODE_DATA_SIZE.

Table 24-2OSTask structure fields

Field	Description
dram_stack	Pointer to DRAM matrix stack; should be initialized to 0 for audio tasks and to memory region of size SP_DRAM_STACK_SIZE8 bytes.
dram_stack_size	DRAM matrix stack size in bytes; should be initialized to 0 for audio tasks or SP_DRAM_STACK_SIZE8 for graphics tasks.
output_buff	Pointer to output buffer. The "_dram" versions of the graphics microcode will route the SP output to DRAM rather than to the DP. When this microcode is used, this should point to a memory region to which the SP will write the DP command list.
output_buff_size	Pointer to store output buffer length. The SP will write the size of the DP command list in bytes to this location.
data_ptr	SP command list pointer. For graphics tasks, this is the application constructed display list. For audio tasks, this command list is created by alAudioFrame(3P).
data_size	Length of SP command list in bytes.

1157319

Table 24-2OSTask structure fields

Field	Description
yield_data_ptr	Pointer to buffer to store saved state of yielding task. If the application is going to support preemption of graphics tasks, the graphics tasks should have this structure member set. This should point to a memory region of size OS_YIELD_DATA_SIZE bytes. If task preemption is not supported by the application, this field be initialized to 0. Audio tasks should always set this field to 0
yield_data_size	Size of yield buffer in bytes. When task yielding is to be supported by the application, this should be initialized to OS_YIELD_DATA_SIZE for the graphics task. This should always be 0 for audio tasks.

Note: Refer to the osSpTaskLoad man page for information about the alignment restrictions of the data pointers.

Sending Tasks to the Scheduler: osScGetTaskQ()

Once you have created and initialized a Scheduler task, you can send it to the Scheduler thread via the Scheduler's task queue. You can obtain a pointer to this queue by calling osScGetTaskQ().

The Scheduler will read this task queue after the next retrace message from the Vi Manager. Normally, you will send one audio and one graphics task to the Scheduler each frame.

Note: After you send the task to the Scheduler, you should not modify it until you receive the "done" message.

1157319

PART

Ultra 64 Development Tools

11573189

11573189

Chapter 25

GameShop Debugger

11513189

This chapter describes the game debug environment for the Nintendo Nintendo 64 system. It briefly explains the hardware and software environments, illustrates recommended programming model, tells you how to get started with the debug environment, and introduces you to the most commonly used debugger features.

Hardware Environment

For the development system, the ROM on the game cartridge is replaced by RAM on the development board; in this chapter, we refer to it as "virtual ROM." This allows the game developer to load the game program into memory, control its execution, and observe the effects of modifying the game without having to rebuild from source.

The development board plugs into the GIO bus of the workstation. Audio and video output connections are provided. Communication facilities between the workstation (referred to as the host in the rest of this chapter) and the development board (called the target) are via the RAM devices that emulate the cartridge ROM and several registers provided for handshaking and synchronization.

Software Environment

The software debug environment consists of a number of software modules that must be present to support debugging. Some of these will also be present in the final game system, but many will not. A good understanding of the software architecture will enable the game developer to deal with unexpected situations that arise during a debugging session.

At the highest level, the debugger consists of two major parts. On the development host, a graphically oriented source-level debugger called gvd is provided. In the target system, a small in-circuit debug monitor called rmon acts as the agent for gvd. The operator of the debugger sees only gvd, but requests are actually fulfilled by rmon. That is, you may open a window on the host for the purpose of looking at memory contents. The host cannot access such memory directly, but it can ask rmon to fetch the memory contents from the target so that the host can display them. rmon runs as three threads under the OS, but these threads spend most of their time either blocked (awaiting a host request) or stopped. Thus, they do not interfere with the operation of the game (other than taking up some memory) unless they are processing debugging commands under operator control.

Like the OS and other library routines, rmon is included in a build only if the game developer specifically asks for it. This is done by creating a thread with rmonMain specified as the function to be started when that thread is run. The rmon program is part of libultra, the Nintendo 64 run-time library. You do not need to have any special files to include rmon in a build. Referencing rmonMain automatically includes all code and data for all three of rmon's threads.

On the host side, the main program you see is gvd, the debugger. However, there are a number of support programs that run in conjunction with the debugger. Since gvd is designed to work in other environments as well, it uses a separate program called dbgif (for debugger interface) to communicate with the target environment. Only dbgif knows the actual means of communication with the target system; gvd is independent of such concerns.

Since we wish to share the GIO interface between the host and target with other programs (for example, diagnostics), a third module is provided on the host. This is a device driver built into the UNIX kernel, and functions as the target manager. When any program (such as dbgif) wishes to communicate with the target, it issues requests to the u64 device driver. In this way, it is possible for two pairs of programs running on the host and target to communicate through a single channel without interference.

115-99

Rmon Theory of Operation

As mentioned in the previous section, rmon consist of three threads that run under the operating system, but these threads run very infrequently. The rmon main thread consists of a command parser, a command dispatcher, and a collection of service routines. In operation, the debugger sends a request to the target. This request consists of a number of 32-bit words that describe the work to be done; for example, "read 40 words starting at address 0x10000000 in the address space of thread 6."

Note: All threads run in the same address space in this environment, but the debugger could support a more complex environment where this was not the case. The debugger does consider the RCP to be a separate address space internally.

This request is passed through dbgif to the driver. The host (through operation of the driver) alerts the target that it wishes to send a message. A very small, high-priority thread called the rmon IO thread responds to the interrupts that occur when the driver writes to one of the GIO registers. Only one access to the "virtual ROM" is allowed at a time, so the host must wait until any DMA access in progress is completed.

When this has happened, the target notifies the host that it is now possible to use the memory. At this point, the target system starts a high-priority system thread (the rmon spin thread) that keeps the game from running and starting any more accesses to virtual ROM. Since the game is not accessing this memory, the host is now free to load the request packet into a predetermined location at the high end of memory. When the packet has been deposited in memory, the host notifies the target that a request has arrived. This stops the rmon spin thread. The rmon IO thread notifies the main rmon thread and waits for the next interrupt.

The rmon main thread wakes up in response to the message from the rmon IO thread. It fetches the incoming packet and dispatches a service routine based on what service was requested. In our example, rmonReadMem will be called. This function examines the arguments, reads the memory, and deposits the contents in another section of virtual ROM as part of a reply packet. It then sends an interrupt to the host, alerting it to the arrival of the reply packet in memory. The host responds to this interrupt by copying the reply packet out of virtual ROM and sending another interrupt to the target.

This provides feedback to the target that the host has finished with the reply buffer and the target may use it again.

Most transactions between the host and target follow this model, but there are a few exceptions. It is likely that the target will asynchronously send a packet to the host that is not a reply to a host request. This occurs whenever a breakpoint has been encountered, for example. Both host and target "sign on" when starting, and each has a reply that it sends to the other when such a sign-on is received. The debugger can also process notification that a thread has been created and destroyed. While not currently used, these may be added in the future.

Target-generated interrupts are received by the driver on the host system and routed to processes (for example, dbgif) that have registered that they would like to receive a given set of interrupts. (Interrupts are associated with a six-bit value identifying which interrupt occurred.) Thus, rmon sends a specific interrupt code to the host. This code indicates that the message should be send to dbgif and not some other process. The driver does not read the communication buffers except as an agent for dbgif or another application process.

Programming Model

While a game may use any programming style desired by its author(s), there are certain restrictions imposed by the debugger. Those developers who want to use the debugger must conform to the rules of the programming model to obtain the benefits of source-level debugging. This section discusses the restrictions that apply.

The most obvious requirement is that you must use the OS, since the debugger depends on it. It will not work under an OS of your own design, because it is designed for the Nintendo 64 OS.

Use of the debugger also requires that you restrict thread priorities to a specific range. User threads (those that are part of the game) are assigned the range 1 through 127, with 127 being the highest-priority thread. The OS does not prevent you from assigning thread priorities higher than 127, but you will be unable to debug them. In fact, use of priorities in this range may prevent the debugger from working at all. While the OS does not impose any restrictions on the idlethread (other than the requirement that there be one),

11537569

the debugger requires that the idlethread be assigned priority level zero. It is not sufficient that it be the lowest priority thread in the system: it must be zero. Otherwise, the debugger may attempt to suspend it, which will lock up the system. The rmon main thread should be set to priority OS_PRIORITY_RMON.

The boot procedure for the system is described elsewhere, but some parts of it are repeated here because a review is helpful. Each application has a boot function, which is called at startup (after security checking, of course). The boot function initializes the operating system, and then creates and starts the main thread. The boot procedure may also do other things, such as hardware initialization, if desired. It can also create other threads, but starting a thread is always the last thing the boot procedure does. The reason for this is simple; once control is transferred to a thread, there is no way to get back to the boot procedure. To enable as much debugging of your start-up code as possible, the boot procedure should be minimal—probably just the three function calls that are required to start the main thread.

The main thread starts other threads within the system, including the debugger thread. There is more flexibility here, although the ability to debug system startup is significantly better if the recommended model is followed. The recommended model is for the main thread to create all other threads in the system, start only the rmon thread(s), and then lower its own priority and become the idle thread. Again, you don't have to do this, but debugging will work much better if you do.

Clearly, you can't debug any code that comes before starting the debugger (rmon) thread. It is also the case that you can't really debug code that has already executed by the time the debugger starts up. This is not so much a function of time as it is of the traditional approach used in debugging embedded systems like the Nintendo 64. That is, if you want to watch the system start from inside the debugger, then you can't really start running the application. Since the debugger is just another thread under the OS, it does not keep your application from running off and executing the game application. Some debuggers may "hold off" the application until the debugger is ready; this one doesn't.

Of course, this does not mean that you can't debug the startup of your application. It just means you must bring up your system in a stopped state and start it running from within the debugger. To do this, your code should start only two threads (although it can create as many as it wants, since creating a thread does not cause it to run). The two threads are the rmon thread, which is considered to be only one thread for now, and the idle thread. Comment out or conditionally compile in the osStartThread calls for other threads so that they do not run until told to do so. Running a thread from the debugger is exactly like calling osStartThread.

What happens if you don't follow this procedure and you start all the threads in your system? Unfortunately, in most cases the debugger will be harder to start, since it needs a stopped thread to connect to. The idle thread and the debugger threads will be running, but it is likely that all your application threads will be blocked on some event. Since the OS now allows waiting threads to be stopped, you may bring up the application in a running state, use the multithread view to stop the thread to which you will attach, and then use Switch Thread to connect.

Using the Debugger

Once you have all the required software installed on your system, you can modify your application to include rmon. Since rmon is rather passive, it does not require you to run the debugger. It just waits for incoming requests and does not interfere with the game operation unless requests arrive. An include file, rmon.h, is provided as part of the distribution. It should be included by the file that creates and starts the rmon thread.

Once you have built your application, you are ready to debug it.

1. Start dbgif in a window of its own.
2. Download your application with gload.
3. You may now start gvd itself.

   For the Nintendo 64, it is required that gvd be started with the name of your executable (the boot executable, if there is more than one) on the command line. For example, if your executable is named sample, you would enter:

   ```
   gvd sample &
   ```

   The debugger starts. It makes no attempt to contact the target system yet.

1153389

You should have a source window and a small status window (which may be minimized if desired). Now you must establish a link to the target.

4. Select the Admin pulldown menu and click Switch Thread.

   You will be prompted for the ID of the thread to which you wish to connect. Under the OS, threads do not really have small integer ID's; instead, they are referenced by the address of their thread control blocks. When you created the thread initially, you assigned it an ID for the debugger to use.

5. Specify the ID you assigned to the thread to which you will be attaching.

   You may only attach to a thread that is in a stopped state. If you start the application with all threads stopped as recommended above, you will not have any problems attaching.

Once you have successfully attached, the host and target will communicate to pass information about the system state back and forth. This takes a few seconds, or even longer if you have many threads. Once completed, you may bring up other views as appropriate to your debug session. Open views by selecting the Views pulldown menu and then clicking on the view you wish to see. The most frequently used of these are:

- register view

  This is where you may examine or modify the contents of all R4300 registers (except for some system control registers). Note that these registers apply to the thread to which you are currently attached. Switching threads with this view open refreshes it with the register contents for the new thread. You can only examine and modify the registers of a thread that is stopped.

- memory view

  As you would expect, this is where you examine and modify memory contents. You may specify the window origin by address or symbol. This window has two modes. In single-word mode, it displays and modifies exactly one memory word without touching any other locations. This is the mode you would use for dealing with memory-mapped registers. In block mode, it displays a block of memory from the specified starting address. The size of the block is mostly determined by the size of the window on your screen.

11573189

Stretching the window gives you more memory to look at. Shrinking it gives you less. You may specify the base in which you wish memory to be displayed.

- disassembly view

  This view shows you memory contents as disassembled code based on the current PC value, or else disassembled from some address you specify. The source line corresponding to the disassembled memory is also displayed. There are a number of configuration options for this window that let you customize it to the display that you find most useful.

- trap manager

  This view shows you all breakpoints that are set. Breakpoints also show up in the source and disassembly windows as pink lines. The current PC shows up as a green line.

The source view, which is the main view of gvd, consists of a set of control buttons for running and stopping the selected thread, plus two other windows. The source window (the middle portion of the view) displays the source at the current PC (by default), and tracks the program counter to keep it on screen whenever possible. You may set breakpoints here by clicking in the margin to the left of the line at which you wish to set the breakpoint.

The bottom of the source view is a small command line window where you may enter commands and see the results. The mouse cursor must be in this window to use it. This window is usually used to examine data objects like structures. For example, if you wish to look at a message queue called audioMQ, you can enter `print audioMQ`, and the contents of the structure (including all its members) will be printed. Since the compiler and debugger were designed to work together, the debugger has quite good type information for displaying complex structures like this.

If you plan to use this window much, it is probably a good idea to move the debugger higher on the screen and stretch the bottom down to enlarge the command portion of the view. The default size is a bit small. This window accepts most dbx commands, for those of you familiar with this popular UNIX debugger.

The command window is also useful for setting breakpoints in functions that are not on screen because they are in a different source file. While you

1153189

can always change source files and set a breakpoint, it is more convenient (providing you wish to stop at the start of a function) to use the "stop in" command. If you know that you are trying to isolate a problem in a function called sendDisplayList, then it is probably best to type `stop in sendDisplayList` in the command window, then click Continue. This will run your application until any thread enters the specified function.

Note: Encountering a breakpoint stops all threads with priorities in the user range (1 through 127). In general, coprocessor interrupts are blocked while rmon is running, and CPU interrupts are enabled.

The Admin pulldown menu also contains a few other useful items. First, this is how you exit the debugger. You may also change to a different executable here, but you should then do another *Switch Thread* command. There is a multithread view in this menu, which is useful to have opened if you use more than one thread. It allows you to start and stop threads as a group, and indicates whether a given thread is running or stopped. If stopped, it shows you which function it was executing. It also shows you the name of the thread data structure used in thread system calls.

You will probably find gvd to be fairly intuitive, especially if you have used other source level debuggers. The online help should answer most questions that arise in debugger operation.

1157369

11573189

PART

Ultra 64 Performance Tuning

11573189

11573189

Chapter 26

Performance Tuning Guide

1157318 9

The following sections will discuss

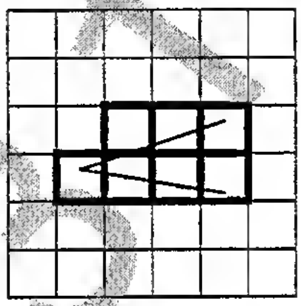
- Data Reduction
- Geometry Tuning
- Raster Tuning
- CPU Tuning

Data Reduction

Game World Organization

The most important performance tuning technique in graphics is to discard as much geometry as possible before animation computation and rendering. Depending on your game, you can organize the geometry in several ways that enable rapid culling of large quantities of data. One example is a simple grid of fixed-sized regions:

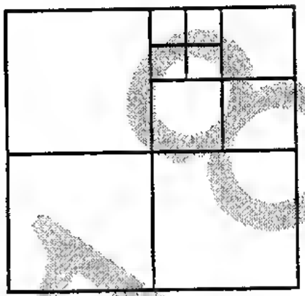
Figure 26-1 Fixed Size Grid Database Organization



11573189

You could also build a hierarchy of different-sized grids to give you a quadtree:

Figure 26-2 Quadtrees

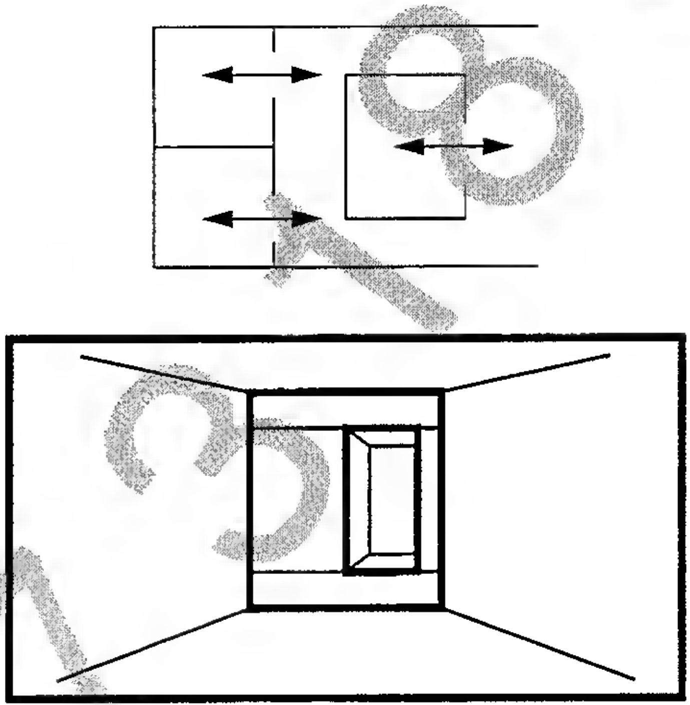


You can extend this into 3D and get either a fixed size cube organization or octrees. Keep in mind that you are trying to eliminate work; not just graphics rendering but also texture loads and animation processing such as collision detection.

The grid need not be regular either, you could also use other boundaries if it suits your data. One example of this is a "portal connectivity" organization inside of a building. In a building with rooms and hallways, the possible list of things that you can see can be represented by a portal connectivity description, which lists which rooms of the building are possibly visible.

You can further reject more data by testing a list of screen projected portal rectangles against visibility to determine whether to consider data in a particular room or hallway.

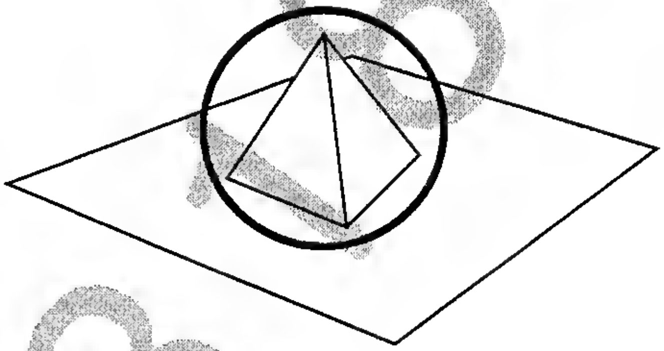
Figure 26-3 Portals Connectivity Visibility



Hierarchical Culling

Throwing away geometry to eliminate processing does not have to stop at the top level. A common organization at the object level is a bounding volume test to eliminate objects (see *gSPCullDisplayList()*).

Figure 26-4 Bounding Sphere Test



1157339

Geometry Tuning (gspFast3D - Precise Microcode)

The standard *gspFast3D* microcode contains very precise subpixel x,y calculations for antialiasing and precise s,t calculations for large screen area textures. This precision is required for terrain or background polygons that are large.

This microcode is full featured, including lighting, clipping, texture coordinate generation (reflection mapping).

Vertex Grouping

The geometry microcode has a local vertex cache. Loading a block of vertexes can amortize the cost of per vertex calculations (transformation, lighting, texture coordinate computation).

Careful organization of the database can minimize these calculations. In general, it is best to load the vertex cache with as many vertices as possible, then draw all the geometry which uses those vertices.

Pre Lighting

For non-dynamic lighting effects, lighting computations can be calculated at model time, then rendered with simple Gouraud shading.

Clipping and Lighting

This microcode does not have enough instruction space to hold lighting and clipping code. It swaps them in from the dram using a least recently used algorithm. Since lighting occurs during vertex load and clipping occurs during polygon drawing, there are natural blocks of work following each ucode load. Loading just a few vertices and then drawing a small number of triangles will cause this microcode loading to "thrash".

1157318

Note: We have not seen performance degradation due to this swap in any games. Game developers did not realize that this was happening until we told them. Large block DMA transfers (such as microcode loads) are very efficient.

Kinds of Polygons

The cost of geometric processing in the RSP is listed below in the order of decreasing performance.

- Flat Shade (using *gDPSetPrimColor (3P)* to select the color)
- Gouraud Shade
- Gouraud Shade + Z-buffer
- Gouraud Shade + Texture
- Gouraud Shade + Z-buffer + Texturing

Textures instead of Geometry

When possible, use textures to represent complex geometry. The RCP is designed to draw high-quality textured primitives. Achieving complexity by using additional geometry will *always* be slower than using textures.

Geometric Level of Detail

When objects get far away or have rapid animation, you can render it with less detail without noticeable loss of detail.

11573189

Geometry Tuning (Turbo Microcode)

The *gspTurbo3D* microcode is a feature-limited, precision-reduced, optimized version of the 3D polygon microcode. It uses a completely different display list organization that is more efficient, but less general.

Because of the reduced precision, the turbo microcode is not suitable for drawing backgrounds or objects with precise textures. It is designed to draw "characters", objects that generally remain in the middle of the viewing frustum.

The following features are not supported with the turbo microcode:

- clipping
- dynamic lighting
- perspective-corrected textures
- matrix stacks
- antialiasing (anti-alising is supported, but not as well).

Current performance measurements of this microcode are >5K polygons per frame @ 60 Hz. For more information, consult the man page for *gspTurbo3D (3P)*.

This microcode is in it's first release and may change.

11573189

Raster Tuning (Fillrate)

Disable Atomic Primitives

Atomic primitive mode (gPipelineMode(G_PM_1PRIMITIVE)) is intended to avoid span buffer coherency problems which can be caused by sucessive primitives with overlapping spans during "read-modify-write" modes (z-buffered or blended modes). The 1PRIMITIVE mode inserts a delay into the pipeline between each primitive to make sure there are no overlaps.

In reality, the overlap case is very rare, and would be hard to see unless you were looking for it. In the worst case, the lost cycles between primitives can add up to about 1-1.5Mpixels/sec of lost fillrate.

To disable the atomic primitive mode, use the command gPipelineMode(G_PM_NPRIMITIVE).

Partial Sorting for Z-Buffer

A "partial sorting" of objects being drawn can accelerate rendering when using z-buffering. The z-buffer test is a conditional write, so if objects are drawn in roughly front-to-back order, this test will often prevent the write to update the z-buffer value.

No Z-Buffer

Z-buffer causes major penalty in fillrate. Antialiasing also causes some performance loss in fillrate. We have included a simple performance tool (blockmonkey) in the release to give you a feel for geometry and fillrate performance.

There are many visibility sorting algorithms available and even more hybrids of these algorithms. There are also properties of particular games that impart valuable information about depth order. If a game can use these techniques and avoid z-buffering, performance will improve.

11513769

Convex Objects

If a group of objects are all convex, a centroid or bounding volume sort and back-face rejection will give the proper rendering order.

Meshed Objects

Many meshed objects have a small number of mesh traversal orders which are correct sorts at arbitrary orientation, even though they are concave. Meshed object are topologically 2D, for example, a torus, a terrain height field, building corridors, etc. With one batch of vertex points, one of several polygon descriptor display lists could be selected by view location. For example, the polygons in a terrain mesh might have four orders across the mesh, S+T+, S-T+, S+T-, S-T-. The two sides of the mesh then closest to the view point select the order.

Cell Based Scenes

Cells are simply a higher level of mesh, where the cell draw order can be determined from view.

Layered Scenes

Often layers of data are known never to be behind another (buildings on a landscape, furniture in a room). then the layers can be drawn in this order, with only a sort within each layer.

Bucket Sort

Attractive since data need only be accessed once. A linked list of buckets can avoid local overflow without excessive memory usage. the bucket can be a display list, for example, of calls to clumps.

Avoid Cyclic Objects

Clumps of polygons in which NO sort order is correct (three long triangles arranged in a triangle in which at each corner a different triangle is in front) have no visibility solution without subdivision.

11513199

Game-Specific Visibility

Many game situations provide implied visibility order between objects or even within objects. Consider a jet fighter flight simulator game. The player is always moving "forward" (in general) and targets attack from a limited number of directions. This could allow you to model the targets carefully and achieve correct surface visibility determination, even if they are not strictly convex.

No Antialiasing

Turning off antialiasing can help increase fillrate. To minimize the aliasing effects, you can increase the horizontal resolution of the framebuffer. Performance tests (blockmonkey) show that 512x240 "no AA no ZB" is faster than 320x240 "AA no ZB" on large polygons. In some cases, this is better than a 25% gain, in exchange for an increase in framebuffer size.

On smaller polygons, you will pay a 5% to 10% fixed overhead due to additional video bandwidth. Both antialiasing and dither filter video hardware require fetching 3 scanlines and filter down to produce a single scanline of video.

Reduced Aliasing

Reduced Aliasing refers to a blender mode (see the G_RM_RA* macros in gbi.h) in which the color and the pixel coverage are only written instead of the normal read/modify /write cycle. In this mode silouette edges will be antialiased, but internal edges of an object will not be antialiased. This mode works with and without z-buffering.

Silouettes can also have artifacts in this mode when displayed on top of a surface which has edges through it, such as a tesselated background, which has also been rendered in this mode. This is because the edges in the background will be partial, rather than fully covered. In this case, the pixel will have multiple partial fragments, and the antialiasing on the silouette will look wrong. A possible workaround for this problem is to render the background in non-antialiased mode, which will write full coverage to the framebuffer. Then render the foreground characters using this reduced antialiasing mode.

CPU Tuning

Parallel Execution of the CPU and the RCP

Full speed rendering in the Nintendo64 can only be accomplished by fully utilizing all of it's resources. One of the most powerful is the coarse-grain parallelism that can be achieved between the CPU and the RCP.

There are many ways you can exploit this parallelism, here are some ideas:

- compute game and animation parameters for frame (n+1) while frame (n) is rendered with the RCP.
- compute game and animation parameters while another RCP task is computing. If your game includes several RCP tasks per frame, you can pipeline them so the CPU and the RCP are always busy at the same time.
- instruct the RDP to render from a DRAM display list while the RSP is used to compute another task, such as audio.

Sorting

A detailed analysis of sorting algorithms is beyond the scope of this document. The reader is referred to texts by Knuth[1] or Sedgewick[2], among others. It is useful to review major properties of sorting algorithm analysis and see how they relate to real-time system performance.

Properties of sorting algorithms which we want to compare include:

- best case sorting time
- worst case sorting time
- average case sorting time

151369

---

[1] Knuth, D. E., *The Art of Computer Programming, Volume 3: Searching and Sorting*, Addison-Wesley Publishing, 1973, ISBN: 0-201-03803-X.

[2] Sedgewick, R., *Algorithms in C*, Addison-Wesley Publishing, 1990, ISBN: 0-201-51425-7.

- additional memory requirements
- size of the code to implement
- ability to exploit coherence.

The time to sort is probably the most important; obviously we want to choose an algorithm that is fast. But it is not that easy. Some of the fastest sorting algorithms have the widest disparity between their average time and their worst-case time. This makes it difficult to predict performance necessary for a real-time system.

Often the difference between worst-average-best-case performance is the initial order of the data. By knowing what we are sorting (and why) we can choose a better sort. For example, if we are sorting Z-values in order to determine visibility drawing order, we can reason that this order varies only slightly from frame to frame (objects do not move "dramatically" and sort interchanges are local). By exploiting this frame to frame coherence, we can choose a sort with linear performance for the "already nearly sorted" case, speeding up our sort tremendously.

Additional memory requirements are also a major concern in an embedded system. They must be minimal, and most of all, predictable. Consider the sorting problem when designing your data structures.

11573789

11573189

Index

11573189

11573189

Symbols
.aiff file 374
.bnk file 426
.ctl file 373, 378, 402, 447, 451
.inst file 76, 397, 449, 451, 457, 458, 459, 462
.sbk file 423, 451, 454
.seq file 451
.sym file 402
.tbl file 402, 426, 451
/usr/sbin 31
/usr/src/PR 30
/usr/src/PR/assets 30
/usr/src/PR/conv 31
/usr/src/PR/libultra 31
/usr/src/PR/relnotes 30
__clearAudioDMA 444
_gsDPLoadTextureBlock_4b 262

Numerics
0x0 122, 139
0x80000400 120
1/w 184, 186
3D transformations 63
4Dgifts 70
64-bit, R4300 46
9-bit RDRAM 318

A
AA_EN 337
a-buffer 340
accuracy, z 325
active page register 58
ADD render mode 344, 345
address 47
ADPCM 369, 373, 385, 401, 402, 405, 412, 413, 414, 426, 435, 455
ADPCM decoder 437
ADPCM decompressor 436
ADPCM predictor 436
ADPCM tools 455
ADSR 406, 430, 457, 458
AI 48, 86, 95, 102, 111, 114
AIFC 76, 412, 413, 435, 451, 455
AIFC spec 435
AIFF 76, 374, 405, 412, 413, 426, 435, 451, 455, 462
AIFF file 459
AIFF-C 405
AL_FX_CUSTOM 388
AL_FX_ECHO 391
AL_FX_SMALLROOM 392
alAudioFrame 65, 372, 382, 383, 395, 469, 470, 475
ALBank 427
ALBankFile 373, 377, 426
alBnkfNew 373, 378, 426
ALCSeq 376
alCSeqGetLoc 377
alCSeqNew 376, 377
alCSeqNewMarker 377
alCSeqNextEvent 377
alCSeqSecToTicks 377
alCSeqSetLoc 377
alCSeqTicksToSec 377
alCSPDelete 379
alCSPGetChlFXMix 380
alCSPGetChlPan 379
alCSPGetChlPriority 380
alCSPGetChlProgram 380
alCSPGetChlVol 380
alCSPGetSequence 379
alCSPGetState 379
alCSPGetTempo 379
alCSPGetVol 379
alCSPNew 379
alCSPPlay 379
alCSPSendMidi 380
alCSPSetBank 379
alCSPSetChlFXMix 380
alCSPSetChlPan 380
alCSPSetChlPriority 380
alCSPSetChlProgram 380
alCSPSetChlVol 380
alCSPSetSequence 379
alCSPSetTempo 379
alCSPSetVol 379
alCSPStop 379
ALDMANew 382
ALDMAproc 382, 383, 384
ALEnvelope 430
alHeapAlloc 447
alHeapInit 372
Alias 70, 71, 72
aliased 271
aliasing 271, 301
alignment 48
alignment, 16-bit 37, 58
alignment, 16-byte 48
alignment, 64 byte 36
alignment, 64-bit 37, 58, 139, 320

alignment, 64-byte 210
alignment, color index palette 244
alignment, image 320
alignment, memory 58
alignment, screen 272
alInit 372, 382, 383, 386
ALInstrument 428
ALKeyMap 431
alpha 287, 332, 336
alpha combiner 291
alpha compare 205, 278, 298, 356
alpha dither 312, 336
alpha times coverage 337
ALPHA_CVG_SEL 337, 338
ALSeq 376
alSeqGetLoc 377
alSeqNew 376, 377, 378
alSeqNewMarker 376, 377
alSeqNextEvent 376, 377
ALSeqpConfig 397
alSeqpDelete 379
alSeqpGetChlFXMix 380
alSeqpGetChlPan 379
alSeqpGetChlPriority 380
alSeqpGetChlProgram 380
alSeqpGetChlVol 380
alSeqpGetSequence 379
alSeqpGetState 379
alSeqpGetTempo 379
alSeqpGetVol 379
alSeqpLoop 380
alSeqpNew 378, 379
alSeqpPlay 378, 379
alSeqpSendMidi 380
alSeqpSetBank 378, 379
alSeqpSetChlFXMix 380
alSeqpSetChlPan 380
alSeqpSetChlPriority 380
alSeqpSetChlProgram 380
alSeqpSetChlVol 380
alSeqpSetSeq 378
alSeqpSetSequence 379
alSeqpSetTempo 379
alSeqpSetVol 379
alSeqpStop 378, 379
alSeqSecToTicks 376, 377
alSeqSetLoc 377
alSeqTicksToSec 376, 377
alSndpAllocate 373, 375
alSndpDeallocate 374, 375
alSndpDelete 374, 375
alSndpGetSound 375
alSndpGetStates 375
alSndpNew 373, 375
alSndpPlay 374, 375
alSndpPlayAt 375
alSndpSetFXMix 375
alSndpSetPan 375
alSndpSetPitch 375
alSndpSetPriority 375
alSndpSetSound 373, 374, 375
alSndpSetVol 375
alSndpStop 374, 375, 458
ALSound 373, 429
alSynAddPlayer 384, 393, 394
alSynAllocFx 393
alSynAllocVoice 384, 393
alSynDelete 393
alSynFreeFx 393
alSynFreeVoice 393
alSynGetFXRef 394
alSynGetPriority 393
alSynNew 382, 393
alSynRemovePlayer 393
alSynSetFXMix 386, 393
alSynSetFXParam 394
alSynSetPan 393
alSynSetPitch 393
alSynSetPriority 385, 393
alSynSetVol 393
alSynStartVoice 385, 393
alSynStartVoiceParams 393
alSynStopVoice 385, 393
ALVoice 384
ALVoiceHandler 395
ALWaveTable 373, 374
ALWavetable 432
ambient 156
animation, sprite 273, 293
antialiasing 46, 63, 74, 119, 175, 203, 204, 207, 301, 302, 327, 340, 342, 343, 356, 496, 498, 501
application thread 33
artifacts, aliasing 271
artifacts, antialiasing 328
artifacts, filtering 274
aspMainDataStart 474
aspMainTextStart 474
attack 374

attack-decay-sustain-release 406, 430
audio 33, 372
audio buffers 442
audio command list 383
audio DAC 41
audio development tools 449
audio DMA callback 383, 470
audio heap 372, 382, 386, 442, 447
audio interface 43, 46, 86, 102
audio library 64, 65, 369
audio playback 52
audio playback rate 382
audio processing 45
audio system 449
audio tools 401
audio waveform 373
Autodesk 3DStudio 71

B
back-face rejection 63, 154, 500
back-facing polygon 329
background image 297
bank 447, 457, 462
bank control file 447
bank file 377, 426, 449, 451, 454
bank object 403
bank, MIDI 30
bilinear filter 193
billboard 205, 262, 286, 332, 333
binary separating planes (BSP) 70
bitmap 354
BL 45, 176, 203, 204, 205
blend 337
blend color 205, 206
blender 45, 203, 301, 305, 310, 317, 327, 331, 345
blender equation 310
blender mode bits, cycle-dependent 345, 346
blender mode bits, cycle-independent 345
blender mode, creation 345
blending 63
blockmonkey 499
blue screen photography 201
Boot 87
boot location 120
bounding volume 495
bounding volume sort 500
box filter 193
breakpoint 93, 486
bss 123

buffers, audio command list 442, 446
buffers, audio output 442
buffers, audio sample DMA 442, 444
buffers, audio sequence 442
buffers, sequence 447
buffers, sequencer event 442, 446
buffers, synthesizer update 442, 446
bus bandwidth 48
byte ordering 425
bzero 119, 123

C
C programming language 38, 47, 58, 67, 77, 137, 457
C, middle C 431, 439, 452
c_dev 30
C3 452
C4 452
cache coherency 55
cache flushing 54
cache invalidate 48
cache line 55, 118
cache line tearing 48
cache, data 118
cache, two-way set-associative 55
cache, vertex 72, 149
cache, write back 118
cached address 128
cached, unmapped 47
CART 95
CaseVision 30
CAUSE register 93
CC 45, 176, 195, 200
cell based scenes 500
centroid sort 500
chroma key 201
CI 190, 215, 221, 290
clamp, coverage 333
CLD_SURF 343, 344, 345
clip ratio 152
clipping 63, 152, 496, 498
clock speed 48
cloud 287, 336
cloud surface 342
cloud surface mode 344
clouds 316
CLR_ON_CVG 330, 337, 338
codebook 436
codecs 65
coherency, span buffer 182

color combiner 45, 193, 195, 200, 278, 288, 291, 295
color combiner input 196
color combiner registers 197
color combiner sources 195
color index 188, 290
color index texture 240
color space conversion 194
command buffer, RDP 109
command list size, audio 446
command list, audio 469
command list, graphics 469
comp.graphics 70
comp.sys.sgi 70
compare, Z 320
compiler, C 77
compiler_dev 30
compressed audio 373
compression 281
Computer Midi Interface 421
computer monitor 74
concave 500
controller input 66
controller interface 86
controllers, sequence player 381
conversion tools 31
convex 501
convex objects 500
coordinate system 146
coprocessor 0, R4300 56
Coprocessor Unusable 93
copy mode 180, 277, 298
copy pipeline mode 276
COUNTER 95
coverage 184, 304, 306, 314, 333, 335, 337, 340, 342
coverage overflow 337
coverage unit 306
coverage value 331, 332
coverage, zap 338
CPU 41, 45, 48, 52, 54, 84, 89, 91, 113, 127, 450, 469, 502
CPU Fault 37
CPU_BREAK 95
cracks 306
culling 492
culling, hierarchical 495
culling, polygon 154
culling, volume 154
CVG_DST 337, 338
CVG_DST_SAVE 317
CVG_X_ALPHA 337, 338
cyclic objects 500

D
DAC 370, 372, 450, 469
data cache, R4300 46, 47, 54, 118, 139
dbgif 31, 67, 480, 481, 482, 484
dbx 486
debugger 67, 90, 93, 124, 479, 480, 481, 482, 484
debugging 37
DEC_LINE 339, 341
decal 295, 337, 343
decal line mode 334, 340
decal surface 332, 333, 334
decay 374
degenerate polygons 331
delta Z 304, 321, 323, 328, 341
depth compare 320
detail texture 229, 230, 233
detune value 459
dev 30
development board 479
development system 48
device driver 101, 480
Device Manager 107
DI 95
diffuse 156
disassembler 37
display list 61, 115, 116, 135, 137, 141, 218
display list, audio 65
display list, optimal 142
display list, RDP 45
dither filter 501
dither, alpha 312
dither, color 210
dither, noise 312
dither, screen coordinate based 312
dithering, color 211
divot 334
DM 107
DMA 37, 44, 46, 48, 54, 55, 56, 58, 101, 112, 114, 139, 383, 445, 470
DMA, audio 445
DMedia 5.5 421
dmedia_eoe (version 5.5) 30
DMEM 44, 115, 135
DP 86, 109, 114
DRAM 60, 63, 239, 475
DRAM, 9-bit 119, 210
dynamic memory allocation 58

1157389

E
effects 386
envelope 373, 377, 402, 406, 457, 458, 461
environment color 197
environment mapping 168
error, Z 325
event 84
example application 384
exception 37, 85, 93
exception handler 85
executable 484
explosions 316

F
far plane 325
fast clears 45
FAULT 34, 95
fault handler 34, 93
file system 87
fill color 211
fill mode 180
FILL_COLOR 352
filter 271
filter, average 276
filter, bilinear 193, 272, 274
filter, bilinear restrictions 193
filter, box 193
filter, point sampling 193
filter, triangular 275
filter, video 314
fixed-point 144, 147, 185, 271
flip, texture 279
floating-point, R4300 46
flt2c 31, 72
fog 169, 179, 203, 205, 206, 313
fog alpha 318
fog color 205
FORCE_BL 317, 337, 338
format, image 318
fractal 234
frame rate, audio 443
FRAME_LAG 445
framebuffer 41, 43, 45, 46, 48, 49, 119, 203, 205, 210, 298
framebuffer alignment 210
framebuffer, color 58
framebuffer, depth 58
frequency, texture 271
FRUSTRATIO_1 152
frustum clipping 63
ftp 70

G
G_AC_DITHER 206, 316, 336
G_AC_NONE 206
G_AC_THRESHOLD 206, 298, 315
G_AD_DISABLE 312
G_AD_NOISE 312
G_AD_NOTPATTERN 312
G_AD_PATTERN 312
G_BL_1 317
G_BL_A_FOG 317
G_BL_CLR_IN 317
G_BL_CLR_MEM 317
G_CC_ADDRGB 198
G_CC_ADDRGBDECALA 198
G_CC_BLENDI 199
G_CC_BLENDIA 199
G_CC_BLENDIDECALA 199
G_CC_BLENDPEDECALA 289
G_CC_BLENDRGBA 199
G_CC_BLENDRGBDECALA 199
G_CC_CHROMA_KEY2 202
G_CC_DECALRGB 198
G_CC_DECALRGBA 198
G_CC_HILITERGB 199
G_CC_HILITERGBA 199
G_CC_HILITERGBDECALA 199
G_CC_INTERFERENCE 200
G_CC_MODULATEI 199
G_CC_MODULATEI_PRIM 199, 288
G_CC_MODULATEI2 200
G_CC_MODULATEIA 199
G_CC_MODULATEIA_PRIM 199
G_CC_MODULATEIDECALA 199
G_CC_MODULATEIDECALA_PRIM 199
G_CC_MODULATERGB 199
G_CC_MODULATERGB_PRIM 199
G_CC_MODULATERGBA 199
G_CC_MODULATERGBA_PRIM 199
G_CC_MODULATERGBDECALA 199
G_CC_MODULATERGBDECALA_PRIM 199
G_CC_PASS2 200
G_CC_PRIMITIVE 198
G_CC_REFLECTRGB 199
G_CC_REFLECTRGBDECALA 199
G_CC_SHADE 198
G_CC_SHADEDECALA 198
G_CC_TRILERP 200

G_CD_BAYER 312
G_CD_DISABLE 312
G_CD_MAGICSQ 312
G_CD_NOISE 312
G_CK_KEY 202
G_CULL_BACK 154
G_CULL_BOTH 154
G_CULL_FRONT 154
G_CV_K0 194
G_CV_K1 194
G_CV_K2 194
G_CV_K3 194
G_CV_K4 194
G_CV_K5 194
G_CYC_1CYCLE 181, 206, 310, 314
G_CYC_2CYCLE 181, 207, 263, 290, 310, 314, 344
G_CYC_COPY 181, 205, 276, 277, 315, 316, 344
G_CYC_FILL 181, 205, 315, 344
G_FOG 169, 207
G_IM_FMT_CI 189
G_IM_FMT_I 189, 288
G_IM_FMT_IA 189
G_IM_FMT_RGBA 189
G_IM_FMT_YUV 189
G_IM_SIZ_16b 189
G_IM_SIZ_32b 189
G_IM_SIZ_4b 189
G_IM_SIZ_8b 189
G_LIGHTING 168
G_MAXFBZ 211
G_MTX_LOAD 145
G_MTX_MODELVIEW 145, 157
G_MTX_MUL 145
G_MTX_NOPUSH 145
G_MTX_PROJECTION 145, 157
G_MTX_PUSH 145
G_OFF 150
G_ON 150
G_PM_1PRIMITIVE 183, 499
G_PM_NPRIMITIVE 183, 499
G_RM_AA_TEX_EDGE 287, 289, 291
G_RM_AA_ZB_OPA_SURF 204
G_RM_AA_ZB_OPA_SURF2 204
G_RM_CLD_SURF 317
G_RM_FOG_PRIM_A 204, 205, 207
G_RM_FOG_SHADE_A 204, 205, 206, 314
G_RM_NOOP 299, 315
G_RM_OPA_SURF 344
G_RM_PASS 204, 205
G_RM_TEX_EDGE 289, 316
G_RM_VISCVG 346
G_RM_VISCVG2 346
G_RM_ZB_CLD_SURF 317
G_RM_ZB_OPA_SURF 299
G_RM_ZB_OPA_SURF2 206
G_TD_CLAMP 192
G_TD_DETAIL 192
G_TD_SHARPEN 192
G_TEXTURE_GEN 168
G_TEXTURE_GEN_LINEAR 168
G_TF_AVERAGE 194, 276
G_TF_BILERP 194, 273, 275
G_TF_CONV 194
G_TF_FILT 194
G_TF_FILTCONV 194
G_TF_POINT 194, 272, 273
G_TL_LOD 192
G_TL_TILE 192, 290
G_TP_NONE 191, 269
G_TP_PERSP 191
G_TT_IA16 192
G_TT_NONE 192
G_TT_RGBA16 192
G_TX_CLAMP 189
G_TX_LOADTILE 225, 248, 292
G_TX_MIRROR 189, 279
G_TX_NOLOD 190, 279
G_TX_NOMASK 189
G_TX_NOMIRROR 189, 279
G_TX_RENDERTILE 225, 248, 273, 275, 276, 292
G_TX_WRAP 189, 283
G_ZS_PRIM 299
gain 377
game controller 29, 43, 46, 112
game timing 55
GameShop 30, 67
gamma correction 74
GBI 61, 62, 188, 216, 218, 248, 351
GBI assembly 62
gbi.h 137, 139, 337, 501
gdis 37
gDPFullSync 36
gDPSetColorImage 35
gDPSetMaskImage 35
gDPSetPrimColor 497
gDPSetTextureImage 35, 216
gDPSetTextureLUT 244, 246
gdSPDefLights0 157

gEndDisplayList 353
General MIDI 467
generation of the MIP maps 232
geometric level of detail 497
geometry 61
ginv 28
GIO 48, 49, 479, 480, 481
GIO board 27
gl_dev 30
gload 31, 34, 37, 78, 87
Gouraud 496
GPACK_RGBA5551 211
GPACK_ZDZ 211
graphics 33
graphics binary interface 61, 62, 72, 137, 216
graphics overrun 471
graphics pipeline 45, 135
gsDPFillRectangle 172
gsDPFullSync 182
gsDPLoadMultiBlock 292
gsDPLoadMultiTile 291, 292
gsDPLoadMultiTile_4b 291
gsDPLoadSync 192, 216, 248
gsDPLoadTextureBlock 163, 166, 216, 225, 262
gsDPLoadTextureTile 189, 248, 282
gsDPLoadTextureTile_4b 189, 288
gsDPLoadTile 216, 225, 248
gsDPLoadTLUT 216, 225
gsDPPipelineMode 183
gsDPPipeSync 181, 311
gsDPSetAlphaCompare 206, 316, 337
gsDPSetAlphaDither 312
gsDPSetBlendColor 311, 315
gsDPSetColorDither 312
gsDPSetCombineKey 202
gsDPSetCombineMode 262, 288, 291
gsDPSetCycleType 169, 181, 206, 263, 276, 277, 310
gsDPSetCyleType 290
gsDPSetDepthSource 299, 309
gsDPSetEnvColor 289
gsDPSetFogColor 169, 205, 207, 311, 313, 318, 344
gsDPSetKeyGB 202
gsDPSetKeyR 202
gsDPSetPrimColor 207, 288, 311
gsDPSetPrimDepth 299, 309, 311
gsDPSetRenderMode 169, 204, 205, 206, 291, 314, 337, 344, 345, 346
gsDPSetScissor 185, 311
gsDPSetTextureConvert 217
gsDPSetTextureDetail 192, 217
gsDPSetTextureFilter 217, 272, 273, 275, 276
gsDPSetTextureImage 248
gsDPSetTextureLOD 192, 217, 290
gsDPSetTextureLUT 216
gsDPSetTexturePersp 191, 216, 269, 270
gsDPSetTile 216, 225, 248, 263
gsDPSetTileSize 216, 225, 248, 263
gsDPTextureRectangle 269, 273, 275, 276, 288
gsDPTextureRectangleFlip 280
gsDPTileSync 192, 216
gsLoadTLUT 191
gSPCullDisplayList 495
gSPDisplayList 35
gSPEndDisplayList 36
gspFast3D 63, 137, 156, 161, 496
gspFast3D_dramDataStart 474
gspFast3D_dramTextStart 474
gspFast3DDataStart 474
gspFast3DTextStart 474
gsPipelineMode 499
gspLine3D 63
gspLine3D_dramDataStart 474
gspLine3D_dramTextStart 474
gspLine3DDataStart 474
gspLine3DTextStart 474
gSPMatrix 35
gSPSegment 138
gSPSetGeometryMode 206
gspTurbo3D 63, 498
gSPVertex 35
gSPViewport 35, 152
gsSetAlphaDither 312
gsSetConvert 194
gsSetFillColor 211
gsSetPrimColor 198
gsSetTextureConvert 194
gsSetTextureFilter 194
gsSetTextureLUT 192
gsSP1Triangle 171
gsSPBranchList 142
gsSPClearGeometryMode 154
gsSPClipRatio 153
gsSPCullDisplayList 154
gsSPDisplayList 141
gsSPEndDisplayList 142, 154
gsSPFogPosition 169, 206, 207
gsSPLine3D 171
gsSPMatrix 145

gsSPPerspNormalize 146, 308
gsSPPopMatrix 145
gsSPSetGeometryMode 154, 168, 169, 206, 207
gsSPSetLights0 159
gsSPTexture 150, 216, 228
gsSPTextureRectangle 172, 216
gsSPTextureRectangleFlip 172, 173
gsSPVertex 149, 160
gsSPViewport 308
guLookAt 144, 152, 163
guLookAtHilite 162
guLookAtReflect 166
guOrtho 144
guParseGbiDL 35
guParseRdpDL 35
guPerspective 144, 146, 152
gvd 31, 34, 67, 87, 124, 480, 484, 486

H
heap library 58
hidden bits 318, 324
high resolution 46
hinv 28
host overrun 470
HW2 interrupt 96

I
I 188, 215, 221, 240, 247, 288
I/O 56, 86, 101, 103
I/O, asynchronous 104
I/O, synchronous 104
IA 188, 215, 221, 240, 247, 289
ic 76, 402, 403, 413, 462
idle thread 33, 90
ie 420
IM_RD 317, 337
image conversion 70
image conversion software 74
image format 318
IMEM 44, 115, 135, 138
immediate mode rendering 61
Indy video input 29
Indy workstation 27, 28, 29, 30, 48, 49, 421
Indy, and MIDI 421
initOsc 397, 398, 399
instruction cache, R4300 46
instrument 376, 377, 398, 404, 427, 429, 457, 461
instrument compiler 362, 402, 403, 412, 435
Instrument Editor 420
integration 33
Intel 425
interference pattern 296
interference texture 261
internal edge 326, 327, 328, 330, 332, 333, 336
interpenetration 303, 337, 338, 342, 343
interpenetration mode 335
interpolation, bilinear 193, 274
interpolation, video filter 326
interrupt 54, 85, 91, 93, 482
interrupt messages 54
inverse kinematics 71
IRIX 30, 67, 77

K
kernel 83
kernel mode 47
keymap 377, 405, 457, 458, 461
Knuth 502
KSEG0 34, 47, 114, 117, 121, 122, 126

L
layered scenes 500
ld 58
level of detail, geometric 70, 497
level of detail, texture 186, 232
libaudio.h 386
libultra 469
libultra.a 31, 77, 78
libultra_d.a 77, 78
light structure 156
lighting 63, 156, 157, 261, 496, 498
line 331
line mode 340
load block 253
load block, line limits 264
load block, restrictions 254
load tile 250
LOD 186, 200, 228, 229, 235
LOD, restrictions 259
log 87
loop 414, 436, 440, 455, 463
loop point 440, 455
low resolution 46

M
M_AUDTASK 474
M_GFXTASK 474
Mach band 211, 312

Macintosh 421
makerom 77, 88, 115, 119, 123, 126
matrix stack 144, 475, 498
matrix stack operations 63
memory allocation 58, 125
memory interface 45, 210, 318
memory management 85, 113
memory map 58
memory, block transfer 250
memory, texture 239
meshed objects 500
message 54, 56, 84, 85, 89, 91, 93
message passing 54
message queue 93, 104, 372, 472
MI 45, 176, 210, 318
microcode, audio 44, 369
microcode, boot 137
microcode, graphics 44, 61, 63
microcode, RSP 43, 45, 47, 60, 137, 216, 469
microcode, task 137
MIDI 30, 64, 79, 369, 376, 378, 401, 402, 403, 407, 416, 423, 454, 457
Midi 421
MIDI file 449, 463
MIDI file format 425
MIDI implementation 449
MIDI key number 405
MIDI message 463
MIDI note 458, 460, 461
MIDI note number 402, 405
MIDI note off 406
MIDI note on 406
MIDI port, Indy 421
MIDI sequence 450
MIDI sequence bank 423
MIDI sequence file 451
MIDI velocities 405
MIDI, compressed 376, 463
MIDI, compressed file format 439
MIDI, standard 376
MIDI, type 0 376
midicmp 75
midicomp 416, 417, 463
midicvt 75, 416, 463
midiDmon 419
midiprint 416
MIP 232
MIP maps, generation 232
mipmapping 150, 179, 184, 223, 229, 232, 291, 333
MIPS R4300 41
mirror, texture 280, 281, 295
mksprite 351
mode, copy 180
mode, decal line 334
mode, fill 180
mode, interpenetration 335
mode, one cycle 177
mode, particle system 336
mode, point sample 338
mode, texture edge 333
mode, two cycle 178
modeling matrix 144
modeling software 70
modulate, color 288
morphing 71, 228, 292
MULTIBIT_ALPHA 262
MultiGen 31, 70
multiple tile effects 261
Music Composition 75
mutual exclusion 105

N
near plane 325
Nichimen Graphics 71
NinGen 70, 72
Nintendo 64 development board 27, 28, 31
NMI 95, 96
noise 302, 312, 337
non-maskable interrupt 96
non-preemptive execution 54
NOOP render mode 344, 345
NTSC 46
NURB 71
Nyquist's Law 271

O
ocean waves 261
octree 493
one cycle mode 177
OPA_DEC 343
OPA_DECAL 339
OPA_INTER 339
OPA_SURF 339, 341, 343, 345
OPA_TERR 339, 341
opaque surface 327, 329, 330, 332, 333, 335, 337, 338, 341
OpenGL 62, 138
operating system 33, 43, 47, 55, 83, 85, 89, 91, 93
OS 480, 482, 484

OS_EVENT_PRENMI 96, 97
OS_K0_TO_PHYSICAL 121
OS_PRIORITY_RMON 483
OS_TASK_DP_WAIT 474
OS_YIELD_DATA_SIZE 476
osAiGetLength 111
osAiGetStatus 111
osAiSetFrequency 111, 372
osAiSetNextBuffer 111, 372
oscDelay 398
oscDepth 398
oscillator 397, 398, 399
osContGetQuery 112
osContGetReadData 112
osContInit 112
osContReset 112
osContStartQuery 112
osContStartReadData 112
oscRate 398
osCreatePiManager 111
osCreateRegion 125
osCreateScheduler 472
osCreateThread 59, 92
osCreateViManager 109
oscState 398
oscType 398
osDestroyThread 92
osDpGetStatus 109
osDpSetNextBuffer 109
osDpSetStatus 109
osFree 126
__osGetCause 98
__osGetCompare 99
__osGetConfig 99
__osGetCurrFaultedThread 34, 100
__osGetFpcCsr 99
osGetIntMask 96
__osGetNextFaultedThread 34, 100
osGetRegionBufCount 126
osGetRegionBufSize 126
__osGetSR 99
osGetThreadId 93
osGetThreadPri 93
osGetTime 55
__osGetTLBASID 99
__osGetTLBHi 99
__osGetTLBLo0 99
__osGetTLBLo1 99
__osGetTLBPageMask 99
osInitialize 88
osInvalDCache 119, 123
osInvalICache 123
osMalloc 125
osMapTLB 127
osPiGetStatus 111
osPiRawReadIo 111
osPiRawStartDma 111
osPiRawWriteIo 111
osPiReadIo 111
osPiStartDma 112
osPiWriteIo 111
osScAddClient 472
osScGetTaskQ 476
OSScTask 473
__osSetCause 98
__osSetCompare 99
__osSetConfig 99
osSetEventMesg 96, 97, 106
__osSetFpcCsr 99
osSetIntMask 96
__osSetSR 99
osSetThreadPri 93
osSetTLBASID 127
osSpTaskLoad 476
osSpTaskStart 109, 383
osSpTaskYield 109, 471
osSpTaskYielded 109
osStartThread 91, 92, 484
osStopThread 93
osSyncPrintf 33, 87
OSTask 137, 383
OSThread 90
osUnmapTLB 127
osUnmapTLBALL 127
osViGetCurrentField 110
osViGetCurrentFramebuffer 110
osViGetCurrentLine 110
osViGetCurrentMode 110
osViGetNextFramebuffer 110
osViGetStatus 109
osVirtualToPhysical 121
osViSetEvent 110
osViSetMode 46, 110
osViSetSpecialFeatures 110
osViSetXScale 110
osViSetYScale 110
osViSwapBuffer 110
osYieldThread 92

output buffer size, audio 446
overlay segments 123
OVL_SURF 343

P
paint software 70, 74
painter's algorithm 340
PAL 46
pan 373, 377, 381, 402, 461
pan values 452
parallel interface 46
particle system mode 336
particle systems 71
PASS render mode 344, 345
patch format 426
PBMPLUS 70
PBUS 49
PC 486
PCL_SURF 339, 341, 343, 345
percussion instrument 406
performance profiling 55
performance tuning 491
performance, CPU 54
peripheral interface 56, 86, 102
peripherial device 43
perspective correction 215, 277, 498
perspective normalization 144
physical address 44, 45, 47, 114, 115, 122, 139
physical voice 384
PI 48, 56, 86, 95, 102, 106, 111, 114
PI manager 46, 56, 86, 90, 95, 111
PIF 46, 102
pinwheel 327, 338, 341
pipeline mode, copy 205, 276
pipeline mode, fill 205, 210
pipeline mode, one cycle 205
pipeline mode, two cycle 187, 200, 203, 228, 232, 244
pitch 402, 405
pixel 46
pixel format, color 210
pixel format, z 210
playback rate 453, 459
player 372
playseq 384, 388, 389
point sample mode 338
point sample, restrictions 259
point sampling 193, 271, 342
polygon fragment 327
polygon rasterization 61, 63
portal connectivity 493
position 402
PRE_NMI_MSG 97
precision, z 308
preemption 54
preemptive 84, 92
PRENMI 95, 96
PRIM_TILE 235
primitive 269, 297
primitive color 197, 288
primitive tile number 228
PRIMITIVE_COLOR 352
priority 381
program crash 38
projection matrix 144
punchthrough 329, 335

Q
quadrication 254
quadtree 493

R
R4000 44, 46, 135
R4300 42, 47, 54, 55, 61, 77, 89, 93, 96, 113, 127, 137, 485
R4300 CPU 46
RAM 373
ramrom 49
rasterization setup 63
rasterizer 45, 184
RCP 41, 48, 49, 55, 60, 61, 65, 94, 102, 113, 135, 301, 351, 383, 388, 426, 469, 497, 502
rcp.h 110, 111
RDP 43, 45, 52, 60, 86, 102, 150, 175, 178, 213, 269
RDP attribute 182
RDP pipeline 178
RDP primitive 182
RDRAM 48, 49, 58, 102, 105, 109, 318, 442
Reality CoProcessor 41, 43, 113
Reality Display Processor 43, 45, 102, 175, 213, 269
Reality Signal Processor 43, 44, 102
real-time scheduling 55
rectangle 45, 184, 269
rectangle, texture 269
reduced aliasing 501
reduction, polygon count 70
reflection mapping 63, 165, 168, 496
region allocation 125
region allocation library 58
region library 86

register, R4300 46
release 374
release notes 30
render mode 303
render mode, visualizing coverage 346
render modes 339, 341, 343, 344, 345
rendering mode 338
rendering order 333, 334, 335, 340, 500
rendering order, for antialiasing 204
RESET 96
retrace message 472
reverb 381
reverb amount 381
RGB, SGI image format 70, 72
rgb2c 72
RGBA 188, 215, 221, 240, 247, 290
RJ-11 29
RM_ADD 317
rmon 33, 34, 67, 95, 480, 481, 484
rmon.h 484
rmonMain 480
rmonPrintf 67, 68
rmonReadMem 481
ROM 58, 77, 105, 373, 383, 402, 426, 450, 453, 479
ROM cartridge 46, 48
ROM image 77
ROM packing 77
RS 45, 176, 184
RSP 34, 43, 44, 45, 47, 52, 60, 61, 102, 135, 206, 372, 450, 454
RSP data memory 44
RSP instruction memory 44
RSP Scalar Unit 44
RSP Vector Unit 44
rspbootTextEnd 474
rspbootTextStart 474

S
s/w 184, 186
sample converter 455
sample rate 459
sample rate, audio 443
sampled sound playback 369, 373
sampling 271
sampling, point 271
sampling, super 303
sampling, unweighted area 303
sbc 423, 438, 463
sbk 75
scaling, rectangle 271
scaling, sprites 294
scheduler 65, 469, 472
scheduler thread 65
scheduler, CPU 54
scheduling, priority 54
scintillate 271
scissor rectangle 185
scissoring 184
scissoring, rectangle 185
scissoring, restrictions 185
scrolling, of rectangles 275
scrolling, texture 286
Sedgewick 502
segment address 34, 44, 121, 127
segment number 121
segment offset 121
segment table 47
segmented address 45, 47, 115, 138, 174
semaphore 85
semitone 459, 460
sequence back compiler 438
sequence bank file 423
sequence bank format 438
sequence buffer 442
sequence data 376, 450
sequence loop point 376
sequence loops 380
sequence playback 376
sequence player 75, 369, 370, 372, 376, 378, 394, 398, 401, 404, 405, 425, 426, 450, 458, 461
sequence, audio 447
sequenced sound 376
sequencer 431
serial interface 46, 102
serial port manager, Indy 421
SETOTHERMODE 174
sgi.com 70
SH 284
sharpened texture 229, 230, 235
SI 48, 95, 102, 114
silhouette 303, 314, 327, 328, 330, 332, 343, 344
silhouette edge 204, 328, 333, 334, 337, 340
simple 384
simple, demo application 65
size, texture 289
SL 284
slide, texture 283
smoke 316
SNES 29, 74, 455

SoftImage 71
sort 330, 500
sorting 298, 330, 502
sorting algorithms 502
sound 457
sound bank 401
sound duration 374
sound effect 64, 450
sound loop point 374
sound pitch 374
sound playback rate 453
sound player 369, 370, 372, 373, 394, 401, 407, 426, 450, 458, 461
sounds, looped 374
sounds, unlooped 374
source file 487
SP 95, 109, 114, 122
SP_BREAK 95
SP_CUTOUT 356
SP_DRAM_STACK_SIZE8 475
SP_EXTERN 357
SP_FASTCOPY 356
SP_FRACPOS 357
SP_HIDDEN 356
SP_SCALE 356
SP_TEXSHIFT 356
SP_TEXSHUF 357
SP_TRANSPARENT 356
SP_UCODE_DATA_SIZE 474
SP_UCODE_SIZE 474
SP_Z 356
span buffer coherency 182, 499
sparkles 336
spClearAttribute 352
spColor 352
spDraw 353, 356, 359
specular 156
specular highlight 161
spFinish 351
spgame 360
spInit 351
spMove 352
sprite 45, 70, 262, 269, 273, 279, 293, 294, 297, 298, 349
sprite library 349
sprites, attribute 352, 355
sprites, bitmap structure 354
sprites, color 352
sprites, creating 351
sprites, cutout 356
sprites, drawing 353
sprites, examples 360
sprites, in COPY mode 356
sprites, moving 352
sprites, re-use 359
sprites, scaling 352, 356
sprites, scissoring 353
sprites, structure 354
sprites, transparent 356
sprites, z-buffered 352
spScale 352
spScissor 352
spSetAttribute 352
spSetZ 352
sptask.h 137
stack overflow 55
stack, thr[illegible] 59
stacktool 446
stereo 46
stipple transparency 336
stopOsc 397, 398, 399
SU 44
SUB_SURF 338, 339, 341
SUB_TERR 340, 342
subpixel 306
subpixel mask 306
Super Famicom 74
Super Nintendo Entertainment System 29
surface types 203
sustain 381
SW1 95
SW2 95
sync command 45
sync, pipe 45
synchronization, of rendering pipeline 181
synthesis driver 369, 370, 382, 394
synthesizer 372

T

t/w 184, 186
tabledesign 76, 412, 462
task 65, 89, 109, 137, 469, 502
task header 137
task list 43, 60, 137
tasks 42, 43
terrain 335, 340, 496
terrain mode 338, 341
TEX_EDGE 317, 332, 339, 341, 345
TEX_INTER 339
TEX_TERR 339, 342

texel 271
texel format 215, 221, 247, 287
texel size 215, 221
texture clamping 224, 255
texture coordinate 150, 215, 219, 236, 269, 284
texture coordinate mask 223
texture coordinate shift 223
texture coordinate transformation 166, 167
texture coordinate, accuracy 260
texture coordinate, automatic generation 156
texture coordinate, bilerp 236
texture coordinate, high 224
texture coordinate, low 224
texture coordinate, point sampled 236
texture coordinate, restrictions 260
texture copy, restrictions 259
texture edge 344
texture edge mode 332, 333
texture engine 186
texture filter 193, 289
texture filter unit 45
texture filter, restrictions 259
texture format 188
texture line 222
texture line stride 222
texture loading 188, 248
texture loading, 4-bit 254
texture loading, block 188
texture loading, tile 188
texture mapping 213
texture memory 45, 214, 239
texture mirroring 222, 223, 255
texture palette 222
texture sampling 191
texture synchronization 192
texture tile 186, 219
texture tile coordinates 219
texture tile descriptor 225, 228
texture tile line padding 250
texture tile restrictions 220
texture tile, multi tile textures 187
texture tile, multiple 261
texture tile, restrictions 187, 260
texture unit 45
texture wrapping 189, 224, 255
texture wrapping (large texture) 251
texture, 4-bit 254
texture, alignment 259
texture, clamped 215
texture, color index 190, 240
texture, color lookup 190
texture, detail 233
texture, high frequency 232
texture, how stored in TMEM 249
texture, interference 261
texture, level of detail 229
texture, load block 253
texture, mirrored 215
texture, quadricated 254
texture, restrictions 259
texture, sharpen 235
texture, wrapped 215
texture, YUV 242
textured rectangle 297
textures, large 297
texure load padding 222
TF 45, 176, 193
TH 284
thread 54, 84, 89, 480
thread ID 485
thread priority 482
thread stacksize, audio 446
thread, audio manager 65
thread, data structure 90, 92
thread, debug 67, 68
thread, game 66, 106
thread, idle 90
thread, priority 90, 92, 93
thread, runnable 91
thread, running 91
thread, scheduler 65
thread, state 90
thread, stopped 91
thread, switch 485
thread, waiting 91
THREAD_STATUS 95
threads 42
tile descriptor 186, 192, 221, 225, 228, 282, 283, 292, 294
tile selection 228
tile, loading 250
tiling, large texture 297
timer 55
timers 87
TL 284
TLB 34, 47, 55, 85, 114, 126
TLB miss 128
TLUT 189, 190, 244, 245, 290, 351, 360
TLUT restrictions 191

TMEM 45, 150, 186, 188, 190, 214, 222, 239, 292, 297, 298, 358
TMEM address 222
Translation Lookaside Buffer 85
translation lookaside buffer 55, 114, 126
translation, rectangle 271
transparency 182, 203, 205, 278, 289, 298, 301, 331, 336
transparent decal surface 342
transparent line 334
transparent lines 332
transparent surface 329, 330, 331, 333, 334, 337, 340, 341, 342
transparent texture 356
tremolo 397, 398
triangle 45, 184
tri-linear interpolation 327
trilinear MIP mapping 229, 233
Tron mode 334
two cycle mode 178
TX 45, 176, 186, 187
type, texture 288

U
ultra 30
ultra64.h 78, 137
union, C 139
UNIX 480, 486
updateOsc 397, 398, 399

V
vadpcm_dec 412, 414, 415
vadpcm_enc 76, 412, 413, 414, 462
vertex 327
vertex buffer 149
vertex cache 496
vertex normal 157
vertex normals 164, 166
vertex transformation 144
vertical retrace 57, 86, 110, 446
VI 48, 57, 86, 95, 102, 109
VI manager 57, 95, 109, 110, 472, 476
VI mode 110
vibrato 397, 398
video filter 314, 326
video interface 43, 46, 86, 102, 110, 328, 334
video mode 46, 57
video retrace 472
video, composite 29, 46
video, RGB 29, 46
video, S-video 29, 46
viewing frustum 498
viewing matrix 144
virtual address 47, 113, 114
virtual ROM 479, 481
virtual voice 384
visibility 494, 499
visibility, game-specific 501
visual complexity 497
voice 384, 395, 453
voice processing estimate 454
voice stealing 385
voice, physical 384
voice, virtual 384
volume 373, 381, 452, 461
VU 44

W
w coordinate 145, 147
waves, ocean 261
wavetable data 402, 405
wavetable file 426
wavetable format 426
wavetable synthesis 64, 369, 414
weather map effect 201
WorkShop 30, 67
wrap, coverage 333, 335, 337, 340
wrap, texture 282, 295

X
XLU_DEC 343
XLU_DECAL 339
XLU_INTER 339
XLU_LINE 331, 339, 341
XLU_SURF 317, 339, 341, 343, 345

Y
yield 60, 84, 89, 109, 476
yield buffer 476
yielding 65
YUV 188, 215, 221, 240

Z
Z compare 320
Z_CMP 337, 338
Z_UPD 337
zap coverage 338, 341
z-buffer 48, 58, 63, 70, 72, 119, 170, 171, 175, 179, 182, 184, 203, 204, 210, 270, 299, 301, 305, 320, 328, 329, 338, 340, 352, 356, 499
z-buffer, alignment 210

z-buffer, format 322
z-buffer, lines 171
ZMODE 337
ZMODE_OPA 317
Z-stepper 308

11573189

NINTENDO 64

Z-Sort Microcode

User's Guide

NUS-06-0164-001A
Released: 1/9/98

D.C.N. NUS-06-0164-001 REV A

"Confidential"

This document contains confidential and proprietary information of Nintendo and is also protected under the copyright laws of the United States and foreign countries. No part of this document may be released, distributed, transmitted or reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from Nintendo.

© 1998 Nintendo

TM® and the "N" logo are trademarks of Nintendo

Table of Contents

Chapter 1 Introduction and Installation 5

Introduction to Z-Sort Microcode 5

Installation 5
Confirm Package Installation 5
For IRIX 5.3, 6.2, 6.3 6
For Partner-N64PC (Windows95/NT) 6

Chapter 2 Z-Sort Microcode Functions 7

Drawing Flow Using Z-Sort 7

Drawing and Arithmetic Operations 8

RSP Processing Installation Methods 8
2-Task Processing 9
2-Pass Parallel Processing 11

Chapter 3 Drawing 13

Drawable Objects (ZObject) 13

ZObject List Processing 13

Z-Sort Processing 14
Preparation of gSPZObject Array 14
Array initialization 15
Array Registration According to Screen Depth of each ZObject 15

ZObject Data Formats 16
zShTri Structure 16
zShQuad Structure 17
zTxTri Structure 20
zTxQuad Structure 21
zNull Structure 23

Controlling RDP Commands with RDPcmd Parameters 23

Clear Screen and Other Drawing Processing 26
gSPZRdpCmd (Gfx *gp. Gfx *rdpcmd) 26

Chapter 4 Arithmetic Operations 27

Display Objects and Arithmetic Operations 27
(Operation A) --- gSPZMultMPMtx 27
(Operation B) --- gSPZLight / gSPZLightMaterial 27
(Operation C) --- gSPZLight/gSPZLightMaterial 27

Work Area for Operations in DMEM 27

GBI List 29

Table of Contents (Continued)

GBI Functions 29
gSPZSetUmem (Gfx *gp, u32 umem, u32 size, u64 *adrs) 29
gSPZGetUMem (Gfx *gp, u32 umem, u32 size, u64 *adrs) 29
gSPZSetUMtx (Gfx *gp, u32 mid. Mtx *mptr) 29
gSPZGetUMtx (Gfx *gp, u32 mid. Mtx *mptr) 30
gSPZMtxCat (Gfx *gp, u32 mids, u32 midt, u32 midd) 30
gSPZMtxTrnsp3x3 (Gfx *gp, u32 mid) 30
gSPZViewPort (Gfx *gp, Vp *vp) 30
gSPZMultMPMtx (Gfx *gp, u32 mid, u32 src, u32 num, u32 dest) 31
gSPZSetAmbient (Gfx *gp, u32 umem, Ambient *ambient); 33
gSPZSetDefuse (Gfx *gp, u32 umem, u32 lid, Light *defuse); 33
gSPZSetLookAt (Gfx *gp, u32 umem, u32 lnum, LookAt *lookat) 34
gSPZXfmLights (gfx *gp, u32 mid, u32 lnum, u32 umem) 34
gSPZLight (Gfx *gp, u32 nsrc, u32 num, u32 cdest, u32 tdest) 35
gSPZLightMaterial (Gfx *gp, u32 msrc, u32 nsrc, u32 num, u32 cdest, u32 tdest) 35
gSPZMixS16 (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor) 36
gSPZMixS8 (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor) 36
gSPZMixU8 (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor) 36
Chapter 5 Other Processing 37
GBI List 37
GBI Functions 37
gSPZSetSubDL (Gfx *gp, Gfx *subdl) 37
gSPZLinkSubDL (Gfx *gp) 37
gSPZSendMessage (Gfx *gp) 37
gSPZWaitSignal (Gfx *gp, zSignal *sig, u32 param) 38
Chapter 6 Compatibility With Other Microcodes 39
About GBIs 39
Common GBIs 39
gSPZSegment (Gfx *gp, u32 seg, u32 base) 39
gSPZPerspNormalize (Gfx *gp, u16 persp) 39
Chapter 7 CPU Support Library 41
Chapter 8 Sample Programs 43
zonetri/ 43
cubes-1 43

Chapter 1 Introduction and Installation

Introduction to Z-Sort Microcode

Z-Sort Microcode was developed to delete obscured screens at the Nintendo 64 (N64) hardware level using a Z-sort. Z-Sort creates screens using a procedure which sorts all the graphics to be displayed on the screen in order of their depth on the screen and then draws them in order from back to front.

The N64 OS/Library supports obscured screen processing using the Z-Buffer. This processing method judges whether or not a graphic is visible on a pixel-by-pixel basis. Compared with Z-Sort, this has the advantage being able to accurately express the relationship before and after the graphic is displayed. On the other hand, access to RAM increases. With Z-Sort, although the relationship before and after display cannot be processed to the same extent as with the Z-Buffer, the amount of RAM access per graphic decreases. Thus, the amount of graphics displayed on the screen within a specific time increases compared to the Z-Buffer method.

The advantage of Z-Sort is that the improved RAM band makes the RDP processing load lighter. In many applications, the time required to perform RDP processing causes a bottleneck. Thus, lighter processing load is ideal when the volume of graphics is high.

One note of caution, however. RSP processing load does not change significantly. RDP processing load changes according to the size of the area to be filled. With a drawing in a small area in particular, RDP processing ends sooner than RSP processing. Because there are many small drawing areas, RDP processing waits for RSP processing to end, during which time the processing capacity does not change with Z-Sort or with Z-Buffer. When the drawing area is somewhat larger, however, the Z-Sort method is effective. Z-Sort Microcode cannot do everything. Carefully consider the screen to be drawn before using Z-Sort.

Installation

This description pertains to installation of Z-Sort Microcode when it is distributed as a separate package. If it is already included in the N64 OS/Library, these operations are not necessary.

Confirm Package Installation

This microcode runs on N64 OS/Library version 2.0H or later. When using 2.0H, confirm that the following packages have been installed. If they are not installed, install them first.

ultra	N64 OS/Library Version 2.0H
patchNmisc_082297	- Patch Nmisc_082297: miscellaneous patches for N64 OS/Library version 2.0H

The Z-Sort package includes the following patch and, therefore, it need not be obtained separately. If the following patch is already installed, install the Z-Sort package as instructed above.

patchNgbi_040997	- Patch Ngbi_040997: patch for gSP1Quandrangle() in gbi.h for N64 OS/Library version 2.0H

For IRIX 5.3, 6.2, 6.3

The Z-Sort Microcode package is formatted as follows.

patchNuZST_mmddyy (mmddyy is the release date)

Install this patch using the Software Manager or the inst command. This will install the following files. For details on the microcode, see the README file.

`/usr/src/PR/doc/gfxucode.Z-Sort/README`	README file
`/usr/lib/PR/gspZ-Sort.fifo.o`	Z-Sort Microcode
`/usr/lib/PR/gspZ-Sort.pl.fifo.o`	Z-Sort Microcode (version with improved arithmetic operations)
`/usr/include/PR/gbi.h`	Z-Sort include file
`/usr/include/PR/gZ-Sort.h`	Z-Sort include file
`/usr/include/PR/rcp.h`	Z-Sort include file
`/usr/src/PR/gZ-Sort/*`	Z-Sort sample programs

For Partner-N64PC (Windows95/NT)

The Z-Sort Microcode package is formatted as follows.

Z-SORTxxx.EXE (xxx is the release number)

This file is self-extracting. When executed, the user will be asked for the installation destination. Input the ROOT directory of the N64 OS/Library. The default is `c:\ultra`. The file opens under the specified directory just as with the IRIX version.

Chapter 2 Z-Sort Microcode Functions

Drawing Flow Using Z-Sort

Z-Sort Microcode supports triangle areas, quadrangle areas, and texture and fill rectangles using RDP commands. In this manual, all areas to be drawn by the RDP are called `ZObjects`.

In Z-Sort Microcode, for each ZObject, one screen depth value is found to represent the drawing area. Each ZObject is then sorted by that screen depth and obscured screen processing is executed by drawing the ZObjects in order from the back to the front.

The processing flow for ZObject drawing is as follows.

1. Multiply model matrix by perspective transformation matrix, etc.
2. Calculate coordinate transformation/perspective transformation/screen depth for model vertices.
3. Determine whether there are vertices in the screen.
4. Determine clipping/back plane.
5. Construct ZObject data.
6. Create ZObject list.
7. Draw in order of ZObject list (drawing processing).

In order to draw a ZObject, the information concerning how the ZObject will be drawn must be prepared as data. With conventional Fast3D Microcode, the Vertex and Tri commands were combined to draw triangles, while with Z-Sort Microcode, drawing is performed by creating ZObject structures.

Not all of these processes are available in Z-Sort Microcode. The major difference between Z-Sort and other graphics microcodes is that Z-Sort Microcode does not function by itself; the CPU must perform some of the processing related with drawing.

For example, the function of sorting ZObjects in order of screen depth is not available as microcode. Since the CPU does not perform sorting, that function must be handed over to the RSP.

At the very least, the CPU must perform the following processes.

- Clipping/back screen determination
- ZObject data construction
- ZObject list creation

Z-Sort Microcode currently offers the following main functions. Each process is controlled by the Display List (DL) comprised of one or more GBI commands.

- Multiplication of model matrix by perspective transformation matrix
- Calculation of coordinate transformation/perspective transformation/ screen depth for model vertices
- Creating flags for whether or not vertices are in the screen
- Drawing in order of ZObject list (drawing processing)

Naturally, matrix multiplication and coordinate transformation (here, called arithmetic operation processing) could also be performed by the CPU. Dividing these tasks between the CPU and the RSP according to available processor capacity is best. For the remainder of the explanation, however, it is assumed that the RSP will perform arithmetic operation processing. If the CPU is to perform operation processing, read about the arithmetic operation processing explained in chapter 4.

Drawing and Arithmetic Operations

When the RSP performs the arithmetic operations, Z-Sort Microcode processing uses two passes. Since the coordinate transformation of the before and after ZObject is not completed, the final Z-Sort results cannot be obtained. This means that data cannot flow in a pipeline like it does with other microcodes. It is necessary to temporarily hold all of the Zobject information.

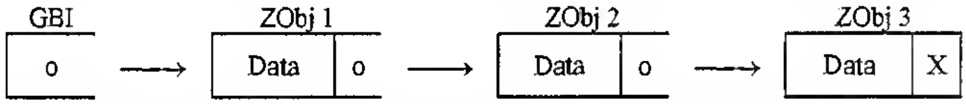
Thus, the following functions related to coordinate transformation are called arithmetic operation processing and are performed on the first pass. These processes are "vertex" coordinate transformations, so ZObject plane data is not created at this time. Note that the CPU creates actual ZObject plane data from the results of vertex coordinate transformation.

- Multiplication of model matrix by perspective transformation matrix
- Calculation of coordinate transformation/perspective transformation/ screen depth for model vertices
- Determination of whether there are vertices in the screen

The next process following Z-Sorting by the CPU, is called "drawing processing" and is performed on the second pass of the RSP.

- Drawing in order of ZObject list

This ZObject list is a chained data string similar to that below, in which ZObject data are linked in the form of a list in order from the back of the screen. The X of ZObj 3 below signifies the end of the chain.



It is necessary that the CPU create this ZObject list. Since Z-Sort Microcode supports ZObjects in list format, the cost of substituting in data when sorting can be kept to a minimum. Any sorting algorithm may be used. Incidentally, in the sample program of this microcode, packet sorting divided into 1024 steps between far and near planes is performed by creating multiple ZObject lists.

Once the above is complete, the processing flow continues as follows.

[CPU] Create arithmetic operation Display List
↓
[RSP] Arithmetic operation
↓
[CPU] Create ZObject data
Create ZObject list (= Display List for drawing) (Z-Sort)
↓
[RSP/RDP] Drawing Processing

RSP Processing Implementation Methods

It was discussed above that the RSP processing is divided into two passes. The methods for implementing this will be explained here.

- Implementation method A) 2-task processing
- Implementation method B) 2-pass parallel processing

A detailed explanation follows. Since A and B each has advantages and disadvantages, select the implementation method carefully.

2-Task Processing

The 2-task processing method starts by dividing the tasks into arithmetic operation processing and drawing processing. This should be an easy method to understand since it resembles the starting methods for other microcodes. The principle must first be understood.

Implementation Method A

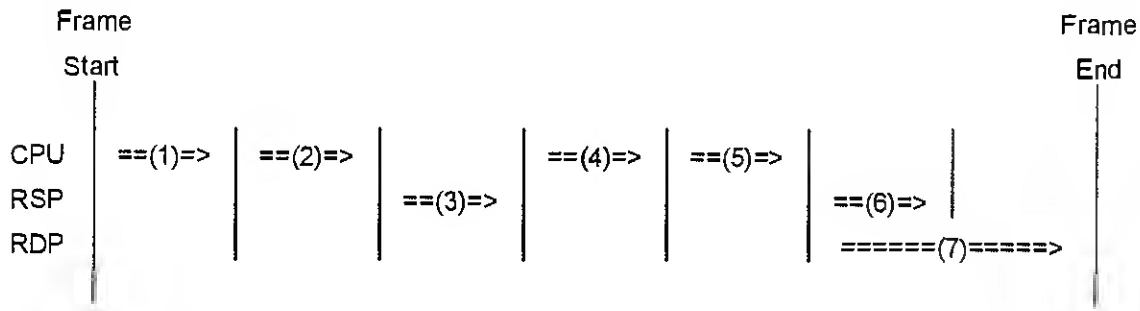
This is the simplest 2-task processing method. It is listed below.

1. Create the Display List for arithmetic operations.
2. Start the first task of the RSP (Display List for arithmetic operations).
3. The RSP performs calculations and the CPU waits until the RSP is done.
4. a. Create ZObject data using the calculation results.
   b. Create ZObject processing links by sorting.
   c. Create the Display List for drawing processing.
5. Start the second task of the RSP (Display List for drawing processing).
6. The RSP performs drawing calculations and the CPU waits until the RDP is done.
7. The RDP performs drawing.

This is the simplest method and, therefore, the easiest to understand. It is effective when shortening the time between key input and screen response. Also, since a single buffer is sufficient as the buffer for developing ZObject data, the amount of memory that should be reserved is decreased.

True of all implementation methods, constructing ZObjects using the CPU in (4) a~c above, requires a considerable computation cost. Differences in the number of ZObjects that can be drawn per frame appear in the ways in which this portion is implementated. If possible, it is recommended that you use "assembly language" instead of C language for this part of the implementation.

The operating status of the CPU, RSP, and RDP for each process is shown below. The numbers in parentheses correspond to those above.

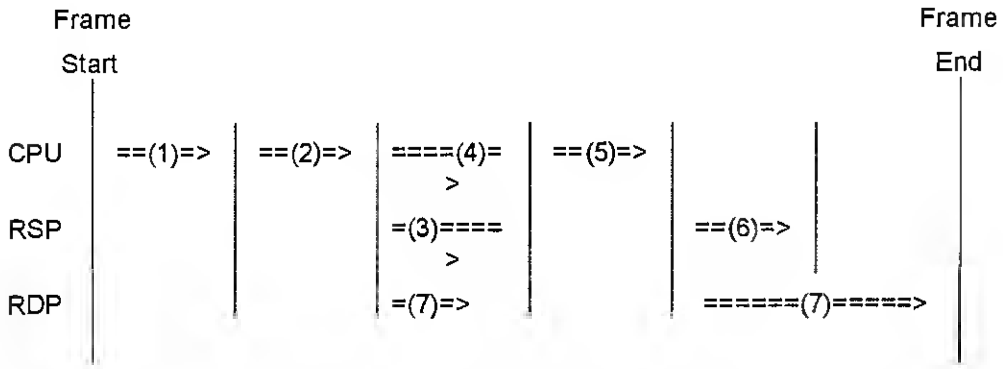


Implementation Method B

One of the problems with implementation method A is that there are no places where the CPU and RSP can operate in parallel. This leaves openings in both CPU and RSP processing. Pipelining processes (3) and (4), in method A, would eliminate some of the space. To create data for a certain number of ZObjects, creation must begin at vertex data points. To support this, Z-Sort Microcode contains a GBI command to send this message to the CPU. When this message is inserted midway through the arithmetic operation processing GBI command, the RSP sends the message to the CPU when the command is processed. When the CPU receives the message, it knows that arithmetic operations prior to the command that sent the message have been completed.

Also, the RDP does nothing during processes (1) to (5) of method A. Thus, the RDP's idle time means reduced drawing performance. Needless to say, to save RDP processing time, it is best that RDP drawing processes that do not require RSP operations, such as screen clearing, be performed within the first RSP pass.

Merely as an example, if the above points are improved, the following results.



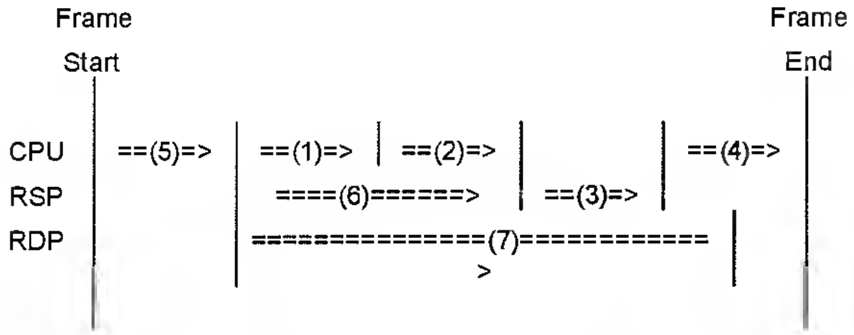
In the third stage, each processor performs the following processing.

CPU:	Creates ZObject data from RSP coordinate calculation data and sorts it. Also, creates the DL for the second pass.
RSP:	Performs coordinate calculations and sends a message to the CPU every time a vertex data point necessary to create a certain amount of ZObject data is obtained.
RDP:	Primarily performs processing that does not require RSP operations, such as screen clearing

Implementating this processing system to perform the above is more complex than system A. There is no significant difference between the difficulty of this processing and that of 2-pass parallel processing described below. Since the performance gain resulting from serial processing (3) and (4) is generally not that great, a different method should be used when reducing the delay in response time after key input and reducing the memory footprint are not important.

Implementation Method C

If the delay in key input response time is acceptable, the following implementation method may be used. The processes (5) through (7) are carried over to the next frame.



In this case, the time between key input and screen response slows, lengthening the RDP processing time.

Since processes (3) and (4) must wait until (6) has ended, the processing time of (6) in the RSP and the processing time of (4) in the CPU must be as short as possible. Since the time the RSP must wait for RDP drawing decreases when the FIFO buffer is enlarged, the processing time of (6) normally shortens, boosting the performance of this processing system. When numerous small ZObjects appear, however, the RSP processing time becomes longer than the RDP's. Since the RDP waits for the RSP, performance does not improve even when the FIFO buffer in enlarged. Thus, it would appear that (4) should be implemented using assembly language.

Considering the ease of implementation and performance, this method appears to be the most balanced among the 2-task processing methods.

Implementation Method D

In the rare event that implementation and sufficient performance in (3) can be obtained using the CPU instead of the RSP, problem-free parallel processing would be possible, as shown below. However, since (6) and (4) sometimes overlap, ZObject data and the DL must be processed using a double buffer.

```
     Start                                                      End
       |                                                         |
CPU    |  ==(5)=>  | ==(1)=> | ==(2)=> | ==(3)=> | ==(4)=>      |
RSP    |           | ====(6)================  |                 |
       |           |                  >       |                 |
RDP    |           | ===============(7)============  |          |
       |           |                      >          |          |
       |                                                         |
```

Whether or not this implementation improves performance depends on the extent to which (3) can be performed faster. If possible, use assembly language for this part of the implementation as was done with (4).

2-Pass Parallel Processing

In graphics processing, the RDP processing time rarely matches the RSP processing time. The FIFO buffer exists to absorb this difference. When the RDP processing time exceeds the RSP processing time, the End Processing RDP command is stored in the FIFO buffer. Since the FIFO buffer size is limited, if the wait is too long, the buffer becomes full.

In other microcodes (Fast3D, F3DEX, S2DEX), when the buffer is full, the RSP waits until space opens up in the FIFO buffer. Merely waiting for RDP processing needlessly consumes the calculation capacity of the RSP.

To eliminate this waste in Z-Sort Microcode, the RSP can perform other DL processing (mainly, arithmetic operation processing) while waiting for RDP processing. This combines arithmetic operation processing and drawing processing into a single task for a pseudo-parallel processing called 2-pass parallel processing.

In 2-pass parallel processing, the DL processed within the RSP stand-by time is called the Sub Display List (Sub DL). Here, as in conventional microcodes, the normal DL is called the Main DL to distinguish it from the Sub DL. Just like the Main DL, the Sub DL has 18 dedicated DL stacks. Since the Sub DL is processed while the RSP is waiting for RDP processing, the GBI commands that can be processed by the Sub DL are limited. Naturally, commands using the RDP cannot be executed. Only commands using the RSP can be used. If GBI commands using the RDP are included in the Sub DL, a malfunction will result. Specific GBI commands which can be included in the Sub DL will be explained later. Mainly arithmetic operation commands can be used.

In actual processing, the RDP processing time usually is not longer than the RSP processing time, and if the RDP drawing area is small, the wasted RSP time mentioned above disappears. When this happens, the Sub DL cannot be processed until expressly called by the Main DL.

The specifications for this microcode assume that there will be inconveniences. Since the RDP drawing area varies depending on the scene to be drawn, the RSP stand-by time in which the Sub DL can be processed is not constant. RSP arithmetic operation processing must end within a certain time to ensure the CPU's ZObject creation time. This is why Sub DL processing even outside the RSP stand-by time is so desirable.

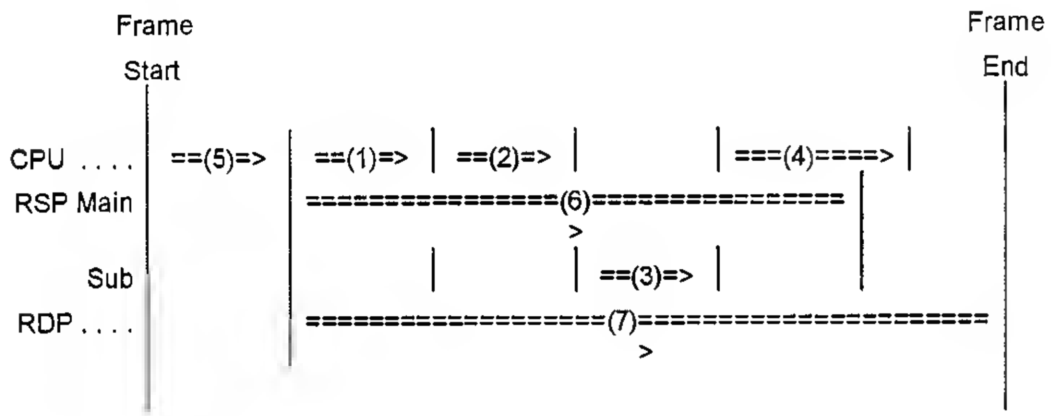
For the above reasons, a microcode `gspZ-Sort.pl.fifo.o` (`Z-Sort.pl ucode`) has been prepared that starts each GBI command in the Sub DL, one at a time, each time a certain amount of ZObject processing is completed; even outside the RSP stand-by time. The timing for calling the Sub DL commands differs depending on the type of ZObject drawn. For polygon ZObjects, one Sub DL command is required for every two to four ZObjects.

In contrast to `Z-Sort.pl ucode`, the microcode `gspZ-Sort.fifo.o` (`Z-Sort ucode`) is for Sub DL processing only during RSP stand-by.

Since this additional processing is performed by `Z-Sort. pl ucode`, the overhead becomes larger than in `Z-Sort ucode`. Therefore, `Z-Sort ucode` offers slightly better RDP drawing performance. These two types of microcode are identical except for the difference in calling the Sub DL and the larger overhead. Select the type desired according to the circumstances.

The 2-pass parallel processing implementation is as follows. Here, (3) and (6) are processed in parallel.

Installation E



Chapter 3 Drawing

Drawable Objects (ZObject)

As explained earlier, in Z-Sort Microcode, graphics are drawn in drawing areas called ZObjects. The drawing parameters for each type of ZObject are defined below, according to the corresponding structure.

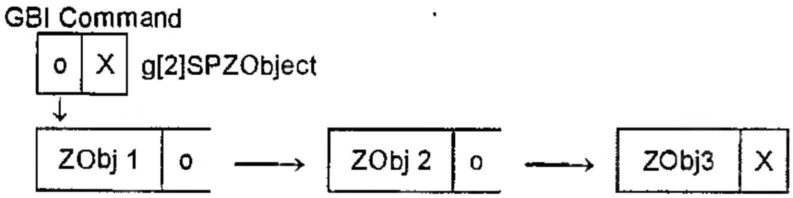
`zShTri`	triangle with smooth shading
`zShQuad`	quadrangle with smooth shading
`zTxTri`	triangle with textured smooth shading
`zTxQuad`	quadrangle with textured smooth shading
`zNull`	other drawing areas using RDP commands (used for Fill Rectangle and Texture Rectangle)

Unfortunately, due to size limitations, Z-Sort Microcode does not provide ZObjects for drawing triangles and quadrangles with flat shading. To draw these, specify the same color for all vertices.

Although the microcode supports only these simple types of graphics, every imaginable type of graphic can be drawn using the libraries in the CPU. For details, refer to the sample programs.

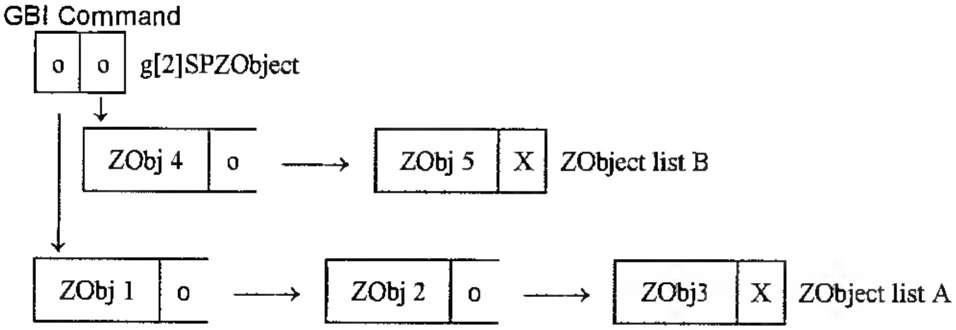
ZObject List Processing

Since ZObjects can be put into a list format, pointer data for the next ZObject and the type ID for the next ZObject can be saved at the head of the structure. The 4 bytes at the head of all ZObject structures are reserved as the header area. ZObjects can be formatted as a list depending on the values of these 4 bytes.



When the pointer and ZObject type ID in the head ZObject (ZObj 1 in the figure above) in the list are specified by the GBI command `g[s]SPZObject`, the RSP draws in order according to this list.

From 0 to 2 lists can be processed by the GBI command `g[s]SPZObject`. In other words, two ZObject lists A and B can be drawn by one GBI command.



The minimum size of a GBI command is 8 bytes, which is equal to two pointer data of 4 bytes each. If fewer than two processing lists are being drawn, write the end value (= `G_ZOBJ_NONE`) in the empty space.

The data format of the GBI command `g[s]SPZObject` is as follows, with the front and back halves being the same.

```
* gSPZObject (Gfx *gp, u32 listA, u32 listB)

      listA Link parameter of ZObject link A = ZHDR (ListA, ZidA)
      listB Link parameter of ZObject link B = ZHDR (ListB, ZidB)
```

31 3	2 0
ListA	ZidA
ListB	ZidB

`ListA/ListB`	The head 8 bits from bit 31 to bit 3 of the pointer to the ZObject list must be 0×80. (Normally 0×80)
`ZidA/ZidB`	The ZObject type ID of head of the ZObject list

`ZHDR` (pointer, type) has been provided as a macro for setting these data (32 bits), and can be used as follows.

```
gSPZObject (gfx, ZHDR (ptr_listA, ZH_SHTRI), ZHDR (ptr_listB, ZH_TXTRI));
```

To change only processing link A or B, the direct value may be substituted in as shown below.

```
*((u32 *) gfx) = ZHDR (ptr_listA, ZH_SHTRI);
```

`ZH_xxxxx` is the ZObject type ID and takes the following five values.

`ZH_SHTRI`	triangle with smooth shading
`ZH_SHQUAD`	quadrangle with smooth shading
`ZH_TXTRI`	triangle with texture map and smooth shading
`ZH_TXQUAD`	quadrangle with texture map and smooth shading
`ZH_NULL`	other drawing areas using RDP commands

Although only `gSPZObject` has been explained here, `gsSPZObject` also exists. Further GBI command explanations follow in later chapters; however, as with this GBI, `gsSPZ***` explanations will be omitted.

Z-Sort Processing

The GBI command `g[s]SPZObject` is a structure listing only the pointer for the ZObject list and type ID of a ZObject. When this command is arrayed in multiple lists, however, three or more ZObject lists can be processed. For each ZObject list, a ZObject list of ZObjects with nearly the same screen depth is created. By listing them in order from the ZObject list at the back of the screen using `g[s]SPZObject`, they can easily be packet sorted.

The processing procedure is as follows. In this example, processing is performed dividing the screen depth for each ZObject into 1024 steps.

Preparation of `gSPZObject` Array

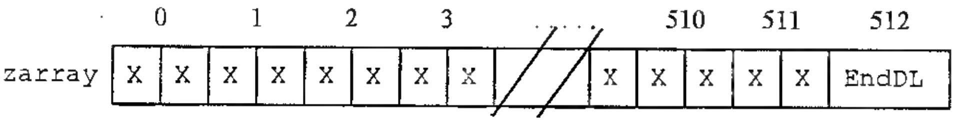
Since there is one ZObject list per screen depth step, 512 commands (=1024/2) are required as the `gSPZObject` array size. Since this array becomes part of the DL and is processed directly, as is, by the RSP, `gSPEndDisplayList` is added to the very end of the `gSPZObject` array. As a result, the required size becomes 513 commands (=512 + 1).

```
|
|      Gfx zarray [1024/2+1]
|
```

Array initialization

Substitute the end value (`G_ZOBJ_NONE = 0x80000000`) to all array elements to initialize the array. Write `EndDL` at the very end, as shown below.

```
      Gfx *zp = zarray + 512;

      gSPEndDisplayList (zp);
      while (zp != zarray) {
            gSPZObject (-- zp, G_ZOBJ_NONE, G_ZOBJ_NONE);
      }
```



X: End value (= G_ZOBJ_NONE)

Array Registration According to Screen Depth of each ZObject

Calculate the screen depth for each ZObject. Although the RSP can calculate the value of the screen depth at each point, the decision as to which value to use as the screen depth for the ZObject is up to the user. Here are some examples of screen depth values.

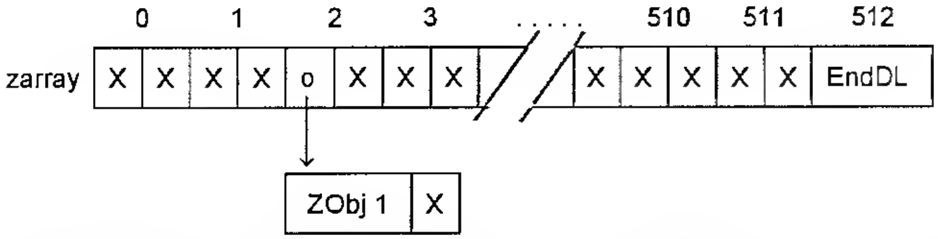
Examples of screen depths in triangle ZObjects

Smallest value for distance from 3 vertices
Largest value for distance from 3 vertices
Average value for distance from 3 vertices
Median value between largest and smallest values for distance from 3 vertices

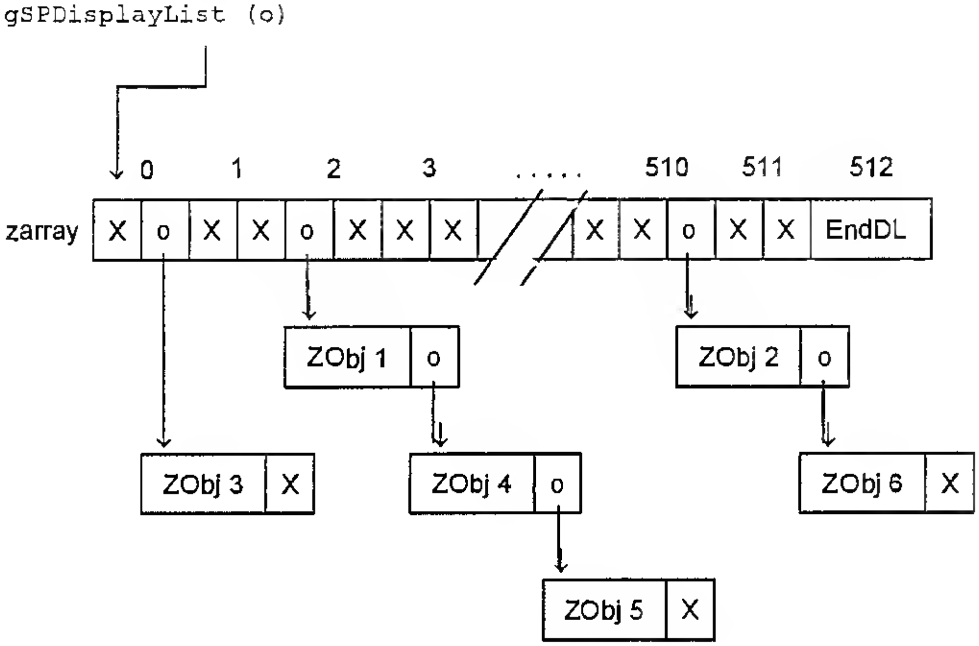
Note: The inverse of the distance can also be used. The sample programs use the average of the inverse values.

This value is normalized between 0 and 1023 and is the number of the array element to register. Store this number in the header of the ZObject in which the pointer and ZObject type ID that originally existed in the applicable array element are registered, and write the pointer to the ZObject structure data and the ZObject type ID to the corresponding array element.

```
      s32    zid;                              /* No. of array element to register
*/
      zHeader *zhptr:                          /* ZObject pointer */
      u32    ztype;                            /* ZObject type ID */

      for ( each ZObject ) {
            Calculate zid from the screen depth;
            if ( zid < 0 )         zid = 0;          /* Clamp zid */
            if ( zid > 1023)       zid = 1023;
            zhptr->t.header = *(uzarray+zid);         /* Set next node */
            *(uzarray+zid) = SHDR (zhptr, ztype);     /* Register to zarray */
      }
```


0
1
2
3
.....
510
511
512
zarray
X
X
X
X
o
X
X
X
X
X
X
X
X
EndDL
ZObj 1
X

Drawing processing can be performed when this process is performed on all ZObjects and the completed arrays are called by `gSPDisplayList`.


gSPDisplayList (o)
0
1
2
3
.....
510
511
512
zarray
X
o
X
X
o
X
X
X
X
X
o
X
X
EndDL
ZObj 1
o
ZObj 2
o
ZObj 3
X
ZObj 4
o
ZObj 6
X
ZObj 5
X

In the above example, drawing is performed in the order ZObj 3 → ZObj 1 → ZObj 4 → ZObj 5 → ZObj 2 → ZObj 6.

ZObject Data Formats

Z-Sort Microcode supports five types of ZObjects and the data required to draw each differs. The five types of structures for storing each type of ZObject data are explained below.

`zShTri` Structure

The `zShTri` structure is used for drawing a triangle with smooth shading and no texture. The following three groups of `zShVtx` vertex data are necessary for specifying this shape.

```
typedef struct {
  s16      x,     y;                  /* Vertex screen coordinates (s10.2) */
  u8       r,     g,     b,     a;    /* Each color in vertex 0. .255 */
}    zShVtx;
```

The `zShTri` structure has the following data format.

	+0		+4			+7
	Hdr		RDP cmd			
V0	X	Y	R	G	B	A
V1	X	Y	R	G	B	A
V2	X	Y	R	G	B	A

```
typedef struct          {
  zHeader               *header;        /* Information on next ZObject */
  Gfx                   *rdpcmd1;       /* Pre-processing DP command */
  zShVtx                v[3];           /* Vertex data */
}       zShTri_t;

typedef struct          {               /* Structure for word access */
  zHeader               *header;
  Gfx                   *rdpcmd1;
  u32                   xy0, clr0;
  u32                   xy1, clr1;
  u32                   xy2, clr2;
}       zShTri_w;

typedef union           {
  zShTri_t              t;
  zShTri_w              w;
  u64                   force_structure_alignment;
}     zShTri;
```

A triangle formed from the three vertices specified by this structure is drawn. At this point, the back side of the triangle is not taken into consideration. The triangle will be drawn regardless of the direction it faces. When a triangle facing the back is not desired, after the CPU determines the front and back when it creates ZObject data, draw only the ZObjects facing the front. The front/back determination is the same as for other polygon ZObjects.

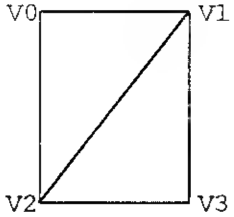
When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject.

The member variable `rdpcmd1` is used to change the current RDP processing mode. Specify the RDP command DL string to be sent to the RDP before drawing the ZObject. For details on `rdpcmd1`, see, “Controlling RDP Commands with `RDPcmd` Parameters” on page 23.

`zShQuad` Structure

The `zShQuad` structure is used for drawing a quadrangle with smooth shading and no texture. The four groups of `zShVtx` vertex data necessary for specifying this shape are given below.

With `zShQuad`, a quadrangle is drawn by drawing the two triangles `V0-V1-V2` and `V1-V2-V3`.



The `zShQuad` structure has the following data format.

	+0		+4			+7
	Hdr		RDP cmd			
V0	X	Y	R	G	B	A
V1	X	Y	R	G	B	A
V2	X	Y	R	G	B	A
V3	X	Y	R	G	B	A

```
typedef struct          {
  zHeader               *header;      /* Information on next ZObject */
  Gfx                   *rdpcmd1;     /* Pre-processing DP command */
  zShVtx                v[4];         /* Vertex data */
}     zShQuad_t;

typedef struct          {             /* Structure for word access */
  zHeader               *header;
  Gfx                   *rdpcmd1;
  u32                   xy0, clr0;
  u32                   xy1, clr1;
  u32                   xy2, clr2;
  u32                   xy3, clr3;
} zShQuad_w;

typedef union           {
  zShQuad_t             t;
  zShQuad_w             w;
  u64                   force_structure_alignment;
}     zShQuad;
```

Memory requirements differ for drawing the same quadrangle using one `zShQuad` function or two `zShTri` functions. Using `zShQuad` requires less memory, a significant advantage.

In addition, RDP drawing performance can be greatly improved by using the CPU to dramatically change the quadrangle's dividing line to better suit RDP drawing. Specifically, compare the absolute value of the Y coordinate of the `V0-V3` diagonal (`ABS Y0-Y3`) to the absolute value of the Y coordinate of the `V1-V2` diagonal (`ABS Y1-Y2`). Then, substitute in the ZObject data so that the quadrangle can be drawn as two triangles along the diagonal with the smaller absolute value. Refer to the following algorithm.

```
zShQuad *zquad;

if (ABS (Y0-Y3) > ABS (Y1-Y2)) {
     /* Divide at diagonal V1-V2, divide into V0-V1-V2 and V1-V2-V3 */
     zquad->t.v[0] = V0; zquad->t.v[1] = V1;
     zquad->t.v[2] = V2; zquad->t.v[3] = V3;
} else {
     /* Divide at diagonal V0-V3, divide into V1-V0-V3 and V0-V3-V2 */
     zquad->t.v[0] = V1; zquad->t.v[1] = V0;
     zquad->t.v[2] = V3; zquad->t.v[3] = V2;
}
```

However, since the diagonal to be selected as the dividing line is unknown at this time, the four specified vertices must be in the same plane so that whichever diagonal is selected, the division of the triangles is problem free. Also, when using texture or smooth shading, the texture coordinate value (`s, t`) or color value (`r, b, g, a`) must be set to avoid contradictions. (A poor example is included in the sample program `cubes-1`.)

For a specific explanation of texture map use. When the vectors `V1, V2,` and `V3` are defined as:

```
V1 = (v1 - v0), V2 = (v2 - v0), V3 = (v3 - v0)
```

in `v0 (x0, y0, z0, s0, t0), v1 (x1, y1, z1, s1, t1), v2 (x2, y2, z2, s2, t2),` and `v3 (x3, y3, z3, s3, t3)`, the actual factors a and b must exist to satisfy:

```
V2 = a * V1 + b * V3.
```

Geometrically, the four vertices in the 5-dimensional coordinate space that included s and t must exist in the same plane.

Similarly when smooth shading and lighting are used, the color value (`r, g, b`) or the normal ray vector (`nx, ny, nz`) must satisfy the above relationship.

For example, the vertices in the `onetri` demo, below, are not good for a quadrangle.

```
onetri/static.c:
  static Vtx shade_vtx[] = {
  {    -64,      64,     -5,      0,       0,       0,         0,  0xff,       0,  0xff },
  {     64,      64,     -5,      0,       0,       0,         0,     0,       0,  0xff },
  {     64,     -64,     -5,      0,       0,       0,         0,     0,    0xff,  0xff },
  {    -64,     -64,     -5,      0,       0,       0,      0xff,     0,       0,  0xff },
  };
```

This part

There would be no problem, however, if this were expressed as illustrated below.

```
static Vtx shade_vtx[] = {
  {    -64,      64,     -5,      0,       0,       0,         0,  0xff,       0,  0xff },
  {     64,      64,     -5,      0,       0,       0,         0,     0,       0,  0xff },
  {     64,     -64,     -5,      0,       0,       0,         0,     0,    0xff,  0xff },
  {    -64,     -64,     -5,      0,       0,       0,      0,  0xff,  0xff,   0xff },
  };
```

In other words, pay close attention when the values between the vertices continuously change. No problem exists with Flat Shading in which the color values between the vertices do not change.

`zShQuad` does not crimp the back of the quadrangle as was done with other ZObjects. Plan for this when the CPU creates ZObject data.

When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject. If there is no next ZObject, substitute the end value `G_ZOBJ_NONE`.

The member variable `rdpcmd1` is used to change the current RDP processing mode. Specify the RDP command DL string to be sent to the RDP before drawing the ZObject. For details on `rdpcmd1`, see, "Controlling RDP Commands with `RDPcmd` Parameters" on page 23.

`zTxTri` Structure

The `zTxTri` structure is for drawing textured triangles with smooth shading. The three groups of `zTxVtx` vertex data necessary for specifying this shape are given below.

```
typedef struct    {
      s16         x, y;        /* Vertex screen coordinates (s10.2) */
      u8          r, g, b, a;  /* Each color in vertex 0. .255 */
      s16         s, t;        /* Texture coordinates in vertex (s10.5) */
      s32         invw;        /* Texture pass vective correction parameter 1/W
                                  (s15.16) (proportion to inverse of distance from
                                   perspective) */
}     zTxVtx;
```

The member variable `invw` is found as shown below from coordinate value `(X, Y, Z, W)` `W` after multiplying the coordinate value of each vertex (`x, y, z, w`) by the MP matrix. However, `perspNorm` is the parameter for normalizing the perspective transformation that can be obtained by the `guPerspective` function.

```
invw = (1<<30)/(perspNorm * W):
```

The RDP uses this value to correct the texture perspective. In the microcode's arithmetic operation processing GBI, this value can be found in the same way as perspective transformation.

The `zTxTri` structure has the following data format.

	+0		+4				+8		+c	+f
	Hdr		RDP cmd 1				RDP cmd 2		RDP cmd3	
V0	X	Y	R	G	B	A	S	T	invW	
V1	X	Y	R	G	B	A	S	T	invW	
V2	X	Y	R	G	B	A	S	T	invW	

```
typedef struct      {
  zHeader           *header;     /* Information on next ZObject */
  Gfx               *rdpcmd1;    /* Pre-processing DP command 1 */
  Gfx               *rdpcmd2;    /* Pre-processing DP command 2 */
  Gfx               *rdpcmd3;    /* Pre-processing DP command 3 */
  zTxVtx            v[3];        /* Vertex data */
}     zTxTri        t;

typedef struct      {            /* Structure for word access */
  zHeader           *header;
  Gfx               *rdpcmd1;
  Gfx               *rdpcmd2;
  Gfx               *rdpcmd3;
  u32               xy0, clr0, st0, invw0;
  u32               xy1, clr1, st1, invw1;
  u32               xy2, clr2, st2, invw2;
}     zTxTri        w;

typedef union       {
  zTxTri_t          t;
  zTxTri_w          w;
  u64               force_structure_alignment
} zTxTri;
```

`zTxTri` does not crimp the back of the triangle as was done with other ZObjects. Plan for this when the CPU creates ZObject data.

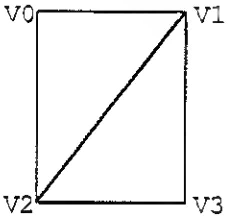
When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject. If there is no next ZObject, assign the end value `G_ZOBJ_NONE`.

The member variables `rdpcmd1, 2,` and `3` are used to change the current RDP processing mode to load the texture. Specify the three RDP command DL strings to be sent to the RDP before drawing the ZObject. For details on `rdpcmd1, 2,` and `3`, see, "Controlling RDP Commands with `RDPcmd` Parameters" on page 23.

`zTxQuad` Structure

The `zTxQuad` structure is for drawing textured quadrangles with smooth shading. The four groups of `zTxVtx` vertex data necessary for specifying this shape are given below.

With `zTxQuad`, a quadrangle is drawn by drawing the two triangles `V0-V1-V2` and `V1-V2-V3`.



The `zTxQuad` structure has the following data format.

	+0		+4				+8		+c	+f
	Hdr		RDP cmd 1				RDP cmd 2		RDP cmd3	
V0	X	Y	R	G	B	A	S	T	invW	
V1	X	Y	R	G	B	A	S	T	invW	
V2	X	Y	R	G	B	A	S	T	invW	
V3	X	Y	R	G	B	A	S	T	invW	

```
typedef struct      {
  zHeader           *header;      /* Information on next ZObject */
  Gfx               *rdpcmd1;     /* Pre-processing DP command 1 */
  Gfx               *rdpcmd2;     /* Pre-processing DP command 2 */
  Gfx               *rdpcmd3;     /* Pre-processing DP command 3 */
  zTxVtx            v[4];         /* Vertex data */
}    zTrQuad_t;

typedef struct      {             /* Structure for word access */
  zHeader           *header;
  Gfx               *rdpcmd1;
  Gfx               *rdpcmd2;
  Gfx               *rdpcmd3;
  u32               xy0, clr0, st0, invw0;
  u32               xy1, clr1, st1, invw1;
  u32               xy2, clr2, st2, invw2;
  u32               xy3, clr3, st3, invw3;
} zTxQuad_w;

typedef union       {
  zTxQuad_t         t;
  zTxQuad_w         w;
  u64               force_structure_alignment
} zTrQuad;
```

For the advantages of using `zTxQuad` and performance enhancing techniques, see the explanation of "`zShQuad` Structure" on page 17.

`zTxTri` does not crimp the back of the triangle as was done with other ZObjects. Plan for this when the CPU creates ZObject data.

When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject. If there is no next ZObject, assign the end value `G_ZOBJ_NONE`.

The member variables `rdpcmd1, 2,` and `3` are used to change the current RDP processing mode to load the texture. Specify the three RDP command DL strings to be sent to the RDP before drawing the ZObject. For details on `rdpcmd1, 2,` and `3`, see, "Controlling RDP Commands with `RDPcmd` Parameters" on page 23.

`zNull` Structure

The `zNull` structure is not for drawing so-called polygons like triangles and quadrangles. It is for drawing rectangle areas drawn by sending direct commands to the RDP (e.g., FillRectangle, TextureRectangle).

Not only the command for drawing the rectangle areas but the type of RDP command can be specified. As a result, a ZObject can be created merely by changing the Fog and Primitive colors and not actually drawing anything.

+0	+4	+8	+c	+f
Hdr	RDP cmd 1	RDP cmd 2	RDP cmd3	

```
typedef struct          {
  zHeader               *header;
  Gfx                   *rdpcmd1;
  Gfx                   *rdpcmd2;
  Gfx                   *rdpcmd3;
} zNull_t

typedef union           {
  zNull_t               t;
  u64                   force_structure_alignment;
} zNull;
```

When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject. If there is no next ZObject, assign the end value `G_ZOBJ_NONE`.

Specify the three RDP command DL strings to be sent to the RDP before drawing the ZObject. For details on `rdpcmd1, 2,` and `3`, see, “Controlling RDP Commands with `RDPcmd` Parameters” on page 23.

Controlling RDP Commands with `RDPcmd` Parameters

Each ZObject structure has one or three `RDP Cmd` areas. The status of the RDP during ZObject drawing processing can be changed by the member variable.

To change the RDP status, use the dedicated DL that lists the GBI commands. This is called the RDP command string.

The RDP command string can contain primarily only commands for controlling the status of the RDP. In other words, the GBI commands that can be used as the RDP command string are limited. The RDP command string and the possible GBIs are shown below. The operation of the GBI commands below is the same as in the Fast3D-compatible microcode. GBI commands not listed below may not work correctly.

GBI Commands Usable in RDP Command Strings

gSPNoOp	gDPNoOp
gSPEndDisplayList	
gDPFillRectangle	
gSPTextureRectangle	gSPTextureRectangleFlip
gDPSetColorImage	gDPSetDepthImage
gDPSetTextureImage	gDPSetScissor
gDPSetFillColor	gDPSetEnvColor
gDPSetFogColor	gDPSetBlendColor
gDPSetPrimColor	gDPSetPrimDepth

gDPSetCombineMode	gDPSetConvert
gDPSetKeyR	gDPSetKeyGB
gDPSetOtherMode	
gDPPipelineMode(*)	gDPSetCycleType(*)
gSPSetTexturePersp(*)	gDPSetTextureDetail(*)
gDPSetTextureLOD(*)	gDPSetTextureLUT(*)
gDPSetTextureFilter(*)	gDPSetTextureConvert(*)
gDPSetCombineKey(*)	gDPSetColorDither(*)
gDPSetAlphaDither(*)	gDPSetAlphaCompare(*)
gDPSetDepthSource(*)	gDPSetRenderMode(*)
gDPSetTile	gDPSetTileSize
gDPLoadBlock	
gDPLoadTextureBlcok	gDPLoadTextureBlockS
gDPLoadTextureBlock_4b	gDPLoadTextureBlock_4bs
gDPLoadTextureBlockYuv	gDPLoadTextureBlockYuvs
gDPLoadMultiBlock	gDPLoadMultiBlockS
gDPLoadMultiBlock_4b	gDPLoadMultiBlock_4bS
gDPLoadTile	
gDPLoadTextureTile	gDPLoadTextureTile_4b
gDPLoadTLUT_pal16	gDPLoadTLUT_pal256
gDPLoadSync	gDPPipeSync
DPTileSync	gDPFullSync

One important note here regarding the inability to use `gSPSegment`. Although the segment address can be used for `gDPSetColorImage`, and the like, the value cannot be set with the RDP command string. Also note that `gSPBranchDL` and `gSPDisplayList` cannot be used.

It is assumed that the three RDP Cmd areas `rdpcmd1`, `rdpcmd2`, and `rdpcmd3` will be used as follows.

rdpcmd1: for setting RDP rendering mode
rdpcmd2: for loading to TMEM (mainly, loading to total TMEM/front half of TMEM)
rdpcmd3: for help in loading to TMEM (mainly, loading to TLUT/back half of TMEM)

Given this assumption, use only `rdpcmd1` for drawing graphics without texture (`zShTri`, `zShQuad`). All three may be specified when drawing textured graphics (`zTxTri`, `zTxQuad`).

Z-Sort Microcode is different from the microcode using the Z Buffer function, in that it draws in order from the back to the front. Thus, it cannot continuously draw only polygons with the same texture. Therefore, when using Z-Sort Microcode, ZObjects must be provided with texture information. However, Z-Sort Microcode is equipped with a mechanism for minimizing the waste that results when a texture that is already loaded to the TMEM is loaded again.

The pointer to the just-processed RDP command string is memorized. This is compared to the pointer to the RDP command string to be processed by the current ZObject and is sent to the RDP only when it is different.

The microcode contains RDP command pointer memory areas for the three RDP commands `rdpcmd1`, `rdpcmd2`, and `rdpcmd3` in DMEM (tentatively called `rdpcmd1_save`, `rdpcmd2_save`, and `rdpcmd3_save`). The algorithm for each process is written on the following page in C language.

For `zShTri, zShQuad` (one RDP Cmd area):

```
if (rdpcmd1 != rdpcmd1_save) {
     Processing of RDP command string displayed by rdpcmd1;
     rdpcmd1_save = rdpcmd1;
}
Drawing of ZObject;
```

The RDP command string for switching to the RenderMode is usually set to `rdpcmd1`. A sample of an RDP command string specific to `rdpcmd1` is given below.

`gsDPSetOtherMode` is the GBI for setting a number of DP mode settings at once. Since many RDP commands can be processed with a single instruction, using this command accelerates the processing speed. The commands marked (*) in the above table of GBI Commands Usable in RDP Command Strings, can be processed collectively by `gDPSetOtherMode`.

```
 #define OTHERMODE_A(cyc)                (G_CYC_##cyc##|G_PM_1PRIMITIVE|G_TP_PERSP|¥
                                         G_TD_CLAMP|G_TL_TILE|G_TT_NONE|G_TF_BILERP|¥

G_TC_FILT|G_CK_NONE|G_CD_DISABLE|G_AD_DISABLE)
 #define OTHERMODE_B(rm)
 (G_AC_NONE|G_ZS_PRIM|G_RM_##rm##|G_RM_##rm##2)

/*------ Shade Triangle mode switching ------*/
 Gfx  modeShTri[] = {
      gsDPPipeSync(),
      gsDPSetOtherMode (OTHERMODE_A (1CYCLE), OTHERMODE_B (RA_OPA_SURF)),
      gsDPSetCombineMode (G_CC_SHADE, G_CC_SHADE),
      gsSPEndDisplayList(),
 };
```

For `zTxTri` and `zTxQuad` (three RDP Cmd areas)

```
if (rdpcmd1 != rdpcmd1_save) {
     rdpcmd1_save = rdpcmd1;
     Processing of RDP command string displayed with rdpcmd1;
}
if (rdpcmd2 != rdpcmd2_save) {
     rdpcmd2_save = rdpcmd2;
     Processing of RDP command string displayed with rdpcmd2;
}
if (rdpcmd3 != rdpcmd3_save) {
     rdpcmd3_save = rdpcmd3;
     if (rdpcmd3 != NULL) {
             Processing of RDP command string displayed by rdpcmd3;
     }
}
```

As with `zShTri` and `zShQuad`, the RDP command string for switching to the RenderMode is set to `rdpcmd1`. A sample of an RDP command string specific to `rdpcmd1` is given below. Palette-switching in the 4b CI texture, etc., can also be included here.

```
/*------ Textured Triangle mode switching ------*/
Gfx  modeTxTri[] = {
     gsDPPipeSync(),
     gsDPSetOtherMode (OTHERMODE_A (1CYCLE), OTHERMODE_B (RA_OPA_SURF)),
     gsDPSetCombineMode (G_CC_MODULATERGB, G_CC_MODULATERGB),
     gsSPEndDisplayList(),
};
```

Set the RDP command for loading the texture to `rdpcmd2`. A sample of an RDP command string specific to `rdpcmd2` is given below.

```
Gfx  modeTxTri2[] = {
     gsDPPipeSync(),
     gsDPLoadTextureBlock       (brick, G_IM_FMT_RGBA, G_IM_SIZ_16b, 32, 32, 0,
                                G_TX_WRAP | G_TX_MIRROR, G_TX_WRAP | G_TX_MIRROR,
                                5, 5, G_TX_NOLOD, G_TX_NOLOD),
     gsSPEndDisplayList(),
};
```

Make settings to `rdpcmd3` the same way as `rdpcmd2`. Although `rdpcmd3` is presumably used for TLUT loading, it can also be used for texture loading.

If `rdpcmd3` is unnecessary, assign `NULL (= 0x00000000)`. At this time, `rdpcmd3_save` is *cleared* by `NULL` and the RDP command displayed by `rdpcmd3` is not processed.

For `zNull` (three RDP command areas):

```
     if (rdpcmd1 != NULL && rdpcmd1 != rdpcmd1_save) {
          rdpcmd1_save = rdpcmd1;
          Processing of RDP command displayed by rdpcmd1;
     }
     if (rdpcmd2 != NULL && rdpcmd2 != rdpcmd2_save) {
          rdpcmd2_save = rdpcmd2;
          Processing of RDP command displayed by rdpcmd2;
     }
     if (rdpcmd3 != NULL && rdpcmd3 != rdpcmd3_save) {
          rdpcmd3_save = rdpcmd3;
          Processing of RDP command displayed by rdpcmd3;
     }
```

There are no particular assumptions regarding `zNull`, so it may be used freely. As can be seen from the above algorithms, when `NULL (=0x00000000)` is set to `rdpcmd1`, RDP commands are not processed. The value of the corresponding `rdpcmd_save` at this time is *saved*.

Note: *Save* at `NULL` specification differs from the `rdp`[illegible] *clear* processing with `zTxTri` and `zTxQuad`.

Clear Screen and Other Drawing Processing

One important note regarding the use of Z-Sort Microcode is the inability to write direct RDP commands to a normal Display List. This is due to its being internally divided into SP command processing and DP command processing. This determines the number of microcode instructions and processing speed.

Normally, background filling processes, such as Clear Screen, are necessary for drawing all ZObjects. In Fast3D-compatible microcodes, such a GBI string is usually created in a static area and is called from the Display List side.

However, since the RDP command string for controlling such DP operations as screen clearing is called from the normal Display List, Z-Sort Microcode contains the following GBI commands. The GBI commands that can be used for the RDP command string are limited, as are which ones can be used during ZObject drawing. Refer to the preceding table. For specific examples, refer to the sample program `cubes-1`.

`gSPZRdpCmd` (Gfx *gp. Gfx *rdpcmd)

This is a pointer to the `rdpcmd` RDP command string.

Process the RDP command string. The RDP commands that can be called, however, are limited. (Refer to the table "GBI Commands Usable in RDP Command Strings" on page 23.)

Chapter 4 Arithmetic Operations

Display Objects and Arithmetic Operations

As explained previously, Z-Sort Microcode, can draw four types of polygons, `zShTri`, `zShQuad`, `zTxTri`, and `zTxQuad`. Though this initially appears to be a small number, many more shapes can be drawn by combining these basic four. This microcode offers the following three principal processing operations.

(Operation A) --- `gSPZMultMPMtx`

```
Model coordinate vertex data +
MxP matrix                    ==> Screen coordinate vertex data
```

(Operation B) --- `gSPZLight / gSPZLightMaterial`

```
Normal ray vector data  +
Material data           +
Light data              +
ModelView matrix        ==> Color data
```

(Operation C) --- `gSPZLight/gSPZLightMaterial`

```
Normal ray vector data        +
Line of sight (LookAt) data   +
ModelView matrix              ==> Texture coordinate (environment map) data
```

In all polygon ZObjects, (Operation A) must be performed to find the screen coordinate vertex data. Also, (Operation B) is required when processing light and (Operation C) is required when processing the environment map.

Each GBI used to perform operations A, B, and C (`gSPZMultMPMtx`, `gSPZLight/gSPZLightMaterial`), however, is insufficient by itself. The vertex data and transformation parameters (matrixes, etc.) must be prepared and the DMEM in the RSP must be loaded before the GBI that performs the operations. In addition, the operation results must be written and returned to the DRAM from the DMEM.

Work Area for Operations in DMEM

Z-Sort Microcode has a GBI for specialized arithmetic operations to perform transformation processing to the 3D model screen coordinate system, lighting calculations, and matrix operations using the RSP.

By combining multiple operations, such values as coordinate and color values necessary to draw ZObjects to the screen can be obtained.

For example, the following GBI commands are combined to transform model coordinates to screen coordinates.

1. gSPZViewPort	Sets VIEWPORT.
2. gSPZPerspNormalize	Sets pass normalization factor.
3. gSPZSetMtx	Loads PROJECTION matrix to work area in DMEM.
4. gSPZSetMtx	Loads MODELVIEW matrix to work area in DMEM.
5. gSPZMtxCat	Multiplies PROJECTION and MODELVIEW matrixes.
6. gSPZSetUMem	Loads model coordinate values inside DRAM to work area in DMEM.
7. gSPZMultMPMtx	Transforms model coordinate values to screen coordinate values.
8. gSPZGetUMem	Outputs screen coordinate values to DRAM.

In Z-Sort Microcode, the work areas used in processing arithmetic operations are stored in DMEM. There are two types of work areas, one for general purpose use and one for matrices, each with the following sizes. Also, the general purpose work area is called the user area.

General purpose work area:		
(User area)	Total	2048 bytes
Matrix work area	Total	192 bytes
(Breakdown) ModelView		64 bytes
Projection		64 bytes
M x P		64 bytes

The user area occupies address 0 to 2047. The application creator determines how this area is to be used.

In `libZ-Sort` of the sample program `cubes-1`, the user area is used as follows. Though the areas overlap, this does not cause a problem because they differ in terms of time sequence. Refer to the user area.

1200-1919:	stores source of model coordinate values	(Can hold up to 120 groups)
0-1919:	stores results of screen coordinate value calculations	(Can hold up to 120 groups)
0-383:	stores source of normal ray vectors	(Can hold up to 128 groups)
512-1023:	stores source of material colors	(Can hold up to 128 groups)
512-1023:	stores results of lighting calculations	(Can hold up to 128 groups)
1024-1535:	stores results of environment texture map coordinate calculations	(Can hold up to 128 groups)
1920-2047:	stores light data	(3 DEFUSE lights + 1 AMBIENT + environment map)

The user can divide up and freely use the user area. Since the matrix area has been prepared for storing matrix data, however, it cannot normally be used for other purposes. The user area can also be divided up in detail by specifying a particular address; in the matrix area, basically one of the `areas (GZM_MMTX, GZM_PMTX, or GZM_MPMTX)` is specified. However, address 0~63 at the head of the user area and address 64~127 can be used for the matrix area. Therefore, the five following areas can be used for matrices. Note that the matrix areas have been named `ModelView/Projection/MxP` for ease of understanding; their functions, however, are identical. If there is any confusion, the `ModelView` matrix can be assigned to the `MxP` matrix area.

`GZM_MMTX`	ModelView matrix area
`GZM_PMTX`	Projection matrix area
`GZM_MPMTX`	MxP matrix area
`GZM_USER0`	User area address 0~63
`GZM_USER1`	User area address 64~127

GBIs used for arithmetic operations operate with either the Main DL or the Sub DL. Thus, pay attention when reading and writing the user area by either DL. When parallel processing by the Main and Sub DLs, Sub DL GBIs sometimes destroy the data calculated by Main DL GBI. Accessing the user area via either DL, therefore, is not recommended. Also, it is better to determine which DL will perform arithmetic operations.

GBI List

This is the list of GBIs for arithmetic operations.

gSPZSetUMem	Writes data to user area
gSPZGetUMem	Reads data in user area
gSPZSetMtx	Writes matrix
gSPZGetMtx	Reads matrix
gSPZMtxCat	Multiplies matrixes
gSPZMtxTrnsp3x3	Inverts 3x3 element of matrix
gSPZViewPort	Sets VIEWPORT
gSPZMultMPMtx	Transforms model coordinate values to screen coordinate values
gSPZSetAmbient	Writes Ambient light (environment light)
gSPZSetDefuse	Writes Defuse light (diffused light)
gSPZSetLookAt	Writes LookAt structure data
gSPZXfmLights	Performs light parameter pre-processing
gSPZLight	Performs light calculations
gSPZLightMaterial	Performs light calculations taking matrix into consideration
gSPZMixS16	Performs s16 numeric interpolation
gSPZMixS8	Performs s8 numeric interpolation
gSPZMixU8	Performs u8 numeric interpolation

GBI Functions

This sections explains the GBIs for arithmetic operations.

`gSPZSetUmem` (Gfx *gp, u32 umem, u32 size, u64 *adrs)

`umem`	`user area address for write destination (0~2040)`
`size`	`write size (8~2048)`
`adrs`	`pointer to write source in DRAM`

This GBI writes data to the user area. `umem` and `size` must be multiples of 8. Also, `adrs` has an 8-byte boundary. If 10 bytes of data are needed, specify 16 bytes.

`gSPZGetUMem` (Gfx *gp, u32 umem, u32 size, u64 *adrs)

`umem`	`user area address for read destination (0~2040)`
`size`	`read size (8~2048)`
`adrs`	`pointer to read destination in DRAM`

This GBI reads data from the user area. `umem` and `size` must be multiples of 8. Also, `adrs` has an 8-byte boundary.

`gSPZSetUMtx` (Gfx *gp, u32 mid. Mtx *mptr)

`mid`	`matrix area for write destination`
`mptr`	`pointer to write source in DRAM`

This GBI writes matrix data in DRAM to the matrix area. Generally, one of `GZM_MMTX`, `GZM_PMTX`, or `GZM_MPMTX` is specified to `mid`. However, the 128 bytes at the head of the user area can also be used. If so, specify `GZM_USER0` and `GZM_USER1`. This allows address 0~63 at the head of the user area and address 64~127 to be used for the matrix area.

`gSPZGetUMtx` (Gfx *gp, u32 mid. Mtx *mptr)

`mid`	`matrix area for write destination`
`mptr`	`pointer to write source in DRAM`

This GBI writes matrix data in DRAM to the matrix area. Generally, one of `GZM_MMTX`, `GZM_PMTX`, or `GZM_MPMTX` is specified to `mid`. However, the 128 bytes at the head of the user area can also be used. If so, specify `GZM_USER0` and `GZM_USER1`. This allows address 0~63 at the head of the user area and address 64~127 to be used for the matrix area.

`gSPZMtxCat` (Gfx *gp, u32 mids, u32 midt, u32 midd)

`mids`	`matrix area S`
`midt`	`matrix area T`
`midd`	`matrix area D`

This GBI calculates `(Matrix D) = (Matrix S) + (Matrix T)`. Generally, one of `GZM_MMTX`, `GZM_PMTX`, or `GZM_MPMTX` is specified to `mids`, `midt`, and `midd`. However, the 128 bytes at the head of the user area can also be used. If so, specify `GZM_USER0` and `GZM_USER1`. This allows address 0~63 at the head of the user area and address 64~127 to be used for the matrix area.

When matrix T and matrix D areas are the same, however, the operation may not perform as expected. There is no problem when areas S and D or S and T are the same.

`gSPZMtxTrnsp3x3` (Gfx *gp, u32 mid)

`mid`	`matrix area to be transposed`

This GBI transposes the 3x3 element of the matrix (`x, y, z`). When the matrix is rotating, the transposed result means the reverse rotation of the source matrix. This transposed matrix is used mainly for light processing.

```
|00  01  02  03|              |00  10  20  30|
|10  11  12  13|      →       |01  11  21  13|
|20  21  22  23|              |02  12  22  23|
|30  31  32  33|              |30  31  32  33|
```

One of `GZM_MMTX`, `GZM_PMTX`, or `GZM_MPMTX` is specified to `mid`. However, the 128 bytes at the head of the user area can also be used. If so, specify `GZM_USER0` and `GZM_USER1`. This allows address 0~63 at the head of the user area and address 64~127 to be used for the matrix area.

`gSPZViewPort` (Gfx *gp, Vp *vp)

`vp`	`pointer to VIEWPORT data`

This GBI is roughly the same as the `gSPViewPort` GBI in F3DEX. Although it sets the VIEWPORT, there are differences in the VIEWPORT data parameters. In Z-Sort Microcode, the parameter to control Fog is specified to the Vp structure member variables `vscale`, `vscale[3]` of `vtrans` and `vtrans[3]` using the following macro,

```
vp->vp.scale[3] = GZ_VIEWPORT_FOG_S (in, out);
vp->vp.trans[3] = GZ_VIEWPORT_FOG_T (in, out);
```

where:

`in:`	Fog start distance
`out:`	Fog end distance

A negative value must be set for the `vscale[1]` value to make the top part of the screen positive, i.e., the right, top, front direction (clockwise system).

Start Fog from a distance of 3000 from the perspective. When specifying so that the background color is uniform at a distance of 4000, initialize as follows.

```
Vp   viewport = {
     SCREEN_WD*2, *SCREEN_HT*2, G_MAXZ/2, GZ_VIEWPORT_FOG_S (3000, 4000)
     SCREEN_WD*2, *SCREEN_HT*2, G_MAXZ/2, GZ_VIEWPORT_FOG_S (3000, 4000)
} ;
```

`gSPZMultMPMtx` (Gfx *gp, u32 mid, u32 src, u32 num, u32 dest)

`mid`	MxP matrix
`src`	user area head address that stores vertex model coordinate values
`num`	number of vertices to be processed
`dest`	head address in user area that stores vertex screen coordinate values after coordinate transformation

This GBI regards the data at the user area's `src` position as the 16-bit x, y, z value. This is multiplied by the 4x4 matrix specified by `mid` and that result (X, Y, Z, W) is normalized by W=1. The screen coordinate value is then obtained by transforming ViewPort to the obtained coordinates. Also, at this time, the flags for the FOG parameter and clipping processing are calculated and that data is output to the `dest` position. Next, 6 is added to `src` and 16 to `dest`, and the process proceeds to the next vertex. The `num` vertices are processed continuously.

The formats of the coordinate values to be input and output at this point are defined as follows as the `zVtxSrc` and `zVtxDest` structures in the header file `gZ-Sort.h`.

```
typedef struct      {
  s16  x, y, z;            /* Vertex model coordinate values (s10.2) */
}     zVtxSrc;             /* Size 6 bytes */

typedef struct      {
  s16  sx, sy;             /* Vertex screen coordinate values (s10.2) */
  s32  invw;               /* Texture pass vective correction parameter 1/W (s15.16)
/*
  s16  xi, yi;             /* X, Y values before normalization (integers only) */
  u8   cc;                 /* Flag for clip processing determination */
  u8   fog;                /* FOG factor */
  s16  wi;                 /* W value (integers only) */
}     zVtxDest;            /* Size total 16 bytes */
```

Since the size of the `zVtxSrc` structure is 6 bytes, pay special attention to the 8-byte alignment when transferring DMA using `gSPZSetUMem`. When the transfer size must be a multiple of 8, the DMA transfer size must be rounded off to a multiple of 8.

Since the size of the `zVtxDest` structure is 16 bytes, only the 128-byte area in the 2048-byte user area can be protected. As a result, the `num` range is from 1 to 128. (In actuality, since light and other processes are performed, the range is usually smaller than this.) At this time, the `num` * 16-byte area from the `dest` address can be rewritten; the exception is when `num` is 3 or less. In this case, the 64-byte area from the position specified by `dest` is overwritten.

For example, when `num` is 3 and `dest` is 0, the correct value after transformation can be stored at address 0~47 and meaningless data can be written to address 48~63. Be careful here because the value of the source of address 48~63 will be destroyed. This specification is necessary for improving the calculation speed.

The routine for this GBI is illustrated below. Be sure that the unprocessed `src` is not overwritten by the `dest` output to allow the `src` and `dest` areas to be overlapped. In `libZ-Sort` of the sample program `cubes-1`, with `src` = 1200~1919 and `dest` = 0~1919, a maximum of 120 vertices can be processed.

```
for (i = 0; i < num; i ++) {
     *dest = MultMP (*src);
     src += 6;
     dest += 16;
}
```

The member variable `invw` is found as shown below from coordinate value (X, Y, Z, W) W after multiplying the coordinate value of each vertex (x, y, z, 1) by the MxP matrix. However, `perspNorm` is the parameter for normalizing the perspective transformation set by `guPerspNormalize`.

```
invw = (1<<30)/(perspNorm * W):
```

The `invw` value can be used, as is, to set `zTxTri/zTxQuad` for the ZObject.

When creating the MP matrix using `guPerspective` and `guLookAt`, the `wi` value usually indicates the distance from the perspective point. Z-Sort can also be performed by selecting this value as the screen depth.

Also, `xi, yi` is the non-normalized coordinate value before perspective transformation. This value can be used mainly for clipping processing. Z-Sort Microcode does not support clipping processing using the microcode. However, clipping can be performed using the `xi, yi, wi` value in the CPU program. The details are explained later in this manual.

By checking the value of the clipping processing determination flag `cc`, it can easily be determined whether that vertex is in ViewPort (visible area). The following explains each `cc` flag.

`GZ_CC_LEFT`	X coordinate is left of Left Plane of visible area
`GZ_CC_RIGHT`	X coordinate is right of Right Plane of visible area
`GZ_CC_TOP`	Y coordinate is above Top Plane of visible area
`GZ_CC_BOTTOM`	Y coordinate is below Bottom Plane of visible area
`GZ_CC_NEAR`	Z coordinate is closer than Near Plane of visible area
`GZ_CC_FAR`	Z coordinate is further from Far Plane of visible area

To determine whether the triangle comprised of the vertices v0, v1, and v2 is completely outside the screen, do an AND for the `cc` value of each vertex as shown below and check to see if the result is 0. If the result is not 0, it means that the entire triangle area is outside at least one of the six clip planes. If this is the case, the processing can be stopped at that point, since the triangle is outside the screen.

```
if (v0. cc & v1. cc & v2. cc) {
     Processing stopped because triangle is outside screen;
}
```

To determine whether the triangle v0, v1, v2 intersects the Near Plane, use the above formula to determine whether the triangle is outside the Near Plane and then perform OR processing. This can be used to determine whether clipping processing is being performed at the Near Plane.

```
if ((v0. cc | v1. cc | v2. cc) & GZ_CC_NEAR) {
     Perform Near clipping processing;
}
```

`fog` is used when performing FOG processing. Using the `fog` value for A in RGBA enables FOG processing. In Z-Sort Microcode, FOG is adjusted by ViewPort's Vp structure parameter. For details, refer to the sample program.

In this GBI, obtainable vertex data is actually used as shown below. The numeric values actually to be assigned to each ZObject structure are the `sx, sy, fog,` and `invw` values. The `invw` or `wi` value can be used as the screen depth value for Z-Sort processing.

```
[ For zShTri structure ]

zVtxDest          *v0, *v1, *v2;
zShTri            *shtri;

/* Screen coordinate setting */
shtri->t.v[0].x = v0->sx;      shtri->t.v[0].y = v0->sy;
shtri->t.v[1].x = v1->sx;      shtri->t.v[1].y = v1->sy;
shtri->t.v[2].x = v2->sx;      shtri->t.v[2].y = v2->sy;

/* The settings below apply only when using Fog */
shtri->t.v[0].a = v0->fog;
shtri->t.v[1].a = v1->fog;
shtri->t.v[2].a = v2->fog;
[ For zTxTri structure ]

zVtxDest          *v0, *v1, *v2;
zTxTri            *txtri;

/* Screen coordinate setting */
txtri->t.v[0].x = v0->sx;      txtri->t.v[0].y = v0->sy;
txtri->t.v[1].x = v1->sx;      txtri->t.v[1].y = v1->sy;
txtri->t.v[2].x = v2->sx;      txtri->t.v[2].y = v2->sy;

/* Texture correction parameter setting */
txtri->t.v[0].a = v0->fog;
txtri->t.v[1].a = v1->fog;
txtri->t.v[2].a = v2->fog;
```

`gSPZSetAmbient` (Gfx *gp, u32 umem, Ambient *ambient);

`gSPZSetDefuse` (Gfx *gp, u32 umem, u32 lid, Light *defuse);

`umem`	head address for light data protection area
`ambient`	pointer to Ambient light structure
`lid`	Defuse light number (0, 1, ...)
`defuse`	pointer to Defuse light structure

These GBIs write Ambient light (environment light) data or Defuse light (planar diffused light) data to the user area. The light data area is protected in advance in the user area. Its size depends on the number of Defuse lights and whether the environment is mapped. It is calculated as follows.

```
(Light data area size) =
     8 + 24 * (number of Defuse lights) + ((environment mapping)? 48 : 0));
```

In `libZ-Sort` of the sample program `cubes-1`, since three Defuse lights and environment mapping are used, the 128 bytes from 1920 to 2047 are reserved for the lights.

Fast3D macros can be used to set the Ambient and Defuse structures. When there are two Defuse lights, they are set using `gdSPDefLights2`, as shown in the example below.

```
/*--- Light parameter ---*/
static Lights2 scene_light =
gdSPDefLights2 (  0x20, 0x20, 0x20,                /* Ambient */
                  0xe0, 0xe0, 0xe0, 0, 40, 80,         /* Defuse 0 */
                  0x40, 0x00, 0x00, 0, 80, 40 );       /* Defuse 1 */

/*--- Load light parameter ---*/
gSPZSetAmbient (gp++, 1920,    &scene_light. a);
gSPZSetDefuse (gp++, 1920, 0, &scene_light. l[0]);
gSPZSetDefuse (gp++, 1920, 1, &scene_light. l[1]);
```

`gSPZSetLookAt` (Gfx *gp, u32 umem, u32 lnum, LookAt *lookat)

`umem`	head address for light data protection area
`lnum`	number of Defuse lights
`lookat`	pointer to LookAt structure

This GBI writes the `LookAt` structure data that constitutes the parameter for environment mapping to the light data area. The light data area is protected in advance in the user area. Refer to the explanation for `gSPZSetAmbient/gSPZSetDefuse` on page 33 for further details.

Z-Sort Microcode supports `TEX_GEN_LINEAR` in the `TEX_GEN` and `TEX_GEN_LINEAR` processing modes of the Fast3D-compatible microcode for environment map processing. It is already set up for `TEX_GEN` processing.

Although the functions `guLookAtReflect` and `guLookAtHilite` in the `gu` library of the N64 OS can be used to set the `LookAt` structure, part of it differs from Z-Sort. Since the macro `guZFixLookAt` is available for correction, correct using this after setting `LookAt` using the library functions.

Shown below is the data write processing when `gdSPDefLights2` is used for two Defuse lights. The lights are set and the reflection is mapped using `guLookAtReflect`.

```
/*--- Light parameter ---*/
static Lights2 scene_light =
gdSPDefLights2 (  0x20, 0x20, 0x20,                        /* Ambient */
                  0xe0, 0xe0, 0xe0, 0, 40, 80,             /* Defuse 0 */
                  0x40, 0x00, 0x00, 0, *80, 40 );          /* Defuse 1 */

/*--- Make reflection parameter ---*/
guLookAtReflect (&dynamicp->viewing, &dynamicp->lookat,
        0, 0, 1000, 0, 0, 900, 0, 1, 0);

guZFixLookAt (&dynamic;*>lookat);

/*--- Load light parameters ---*/
gSPZSetAmbient (gp++, 1920,    &scene_light. a);
gSPZSetDefuse  (gp++, 1920, 0, &scene_light. l[0]);
gSPZSetDefuse  (gp++, 1920, 1, &scene_light. l[1]);

/*--- Load reflection parameters ---*/
gSPZSetLookAt (gp++, 1920, 2, &dynamicp*>lookat);
```

`guZFixLookAt` is defined in `gZ-Sort.h` as shown below.

```
#define guZFixLookAt(lp)
     { (lp)->l[1].l.col[1] = (lp)->l[1].l.colc[1]l = 0x00; }
```

This is because two elements have been cleared to 0. (0x80 has been assigned by the `gu` library.) If you want to optimize your processing time, refer to the source file of the `gu` library function in the N64 OS under the `/libultra/gu` directory in the libultra sample program, to correct and replace the library function.

`gSPZXfmLights` (gfx *gp, u32 mid, u32 lnum, u32 umem)

`mid`	matrix with 3x3 element at upper left of ModelView matrix inverted
`lnum`	number of lights to be processed
`umem`	head address in user area that holds light data

This GBI performs lighting pre-processing. The GBI must be called after one or both of the light data and ModelView matrix has been changed and by the time `g*SPLight` or `g*SPLightMaterial` is called. This enables pre-processing in which light data can be used in light calculations by `gSPZLight` and `gSPZLightMaterial`. Since light data will rarely change in one scene, this GBI is called when the ModelView matrix changes.

To execute this GBI, the reverse rotation matrix of the ModelView matrix is necessary. For this, the matrix with the 3x3 element at upper left of ModelView matrix inverted can usually be used. (The shading is sometimes off when scaling only certain axes, but this is not a notable problem.) `gSPZMtxTrnsp3x3` is used for the inversion.

The number of `lnum` basically is the number of Defuse lights. This does not include the Ambient lights. Also, `lnum` cannot be set to 0 in Z-Sort Microcode. To process only Ambient lighting, specify one black (RGB=0, 0, 0) Defuse light using a dummy.

When processing the environment map, use two so-called Defuse lights. When expressing highlighting and reflection, load the environment map parameter to the light parameter area and set (2 for the number of Defuse lights) to `lnum`.

When using only the environment map without using lights (no Defuse lights), the dummy Defuse light described above is unnecessary. Specify 2 to `lnum` and call the GBI.

Shown below is a processing example with the changed ModelView matrix, the light data area head at address 1920, two Defuse lights, and the environment map.

```
/*--- Set ModelView and MxP matrix ---*/
gSPZSetMtx (gp++, GZM_MMTX, &dynamicp->modeling[i]);
gSPZMtxCat (gp++, GZM_MMTX, GZM_PMTX, GZM_MPMTX);

/*--- Xfm light data ---*/
gSPZMtxTrnsp3x3 (gp++, GZM_MMTX);
gSPZXfmLight (gp++, GZM_MMTX, 1920, 4);
```

`gSPZLight` (Gfx *gp, u32 nsrc, u32 num, u32 cdest, u32 tdest)

`gSPZLightMaterial` (Gfx *gp, u32 msrc, u32 nsrc, u32 num, u32 cdest, u32 tdest)

`msrc`	head address in user area that stores material color data (color of vertices)
`nsrc`	head address of user area that stores normal ray vector data
`num`	number of normal ray vector data to be processed (multiple of 2)
`cdest`	head address of user area that stores color value of vertices after
light	calculation
`tdest`	head address of user area that stores texture and coordinate values
of	vertices after environment map calculation

This GBI regards the data from the `nsrc` address in the user area as the signed 8-bit normal ray vector value (nx, ny, nz). It calculates the lighting using the light parameters specified by `gSPZXfmLight`. This provides the light color that corresponds to the normal ray vectors. The vertex color is obtained by multiplying this light color and the material color, which is the color of the vertex itself, by each r, g, b element. These calculated color values are stored at the `cdest` address in the user area.

With `gSPZLight`, (r, g, b, a) = (255, 255, 255, 255) is used as the material color. As with Fast3D microcode, this indicates vertex coloring using light color. Also, with `gSPZLightMaterial`, use data from the `msrc` address in the user area as (r, g, b, a), in order, as the unsigned 8-bit color data.

In addition, when `LookAt` structure data is set as the light data, lighting calculation and environment map calculation are performed simultaneously. Texture coordinate values (S, T = 0.00~32.00) are output to the `tdest` address in the user area as the calculation results. Even when `LookAt` structure data is not set, an undefined value is output to `tdest` so be careful that the (`num * 4`) bytes area is not destroyed.

After `cdest` and `tdest` are output, 3 is added to `nsrc` and 4 is added to `msrc`, `cdest`, and `tdest`, and the process proceeds to the next vertex. Although the `num` normal ray vectors are processed continuously, `num` must be an even number. If it is odd, (`num+1`) is output to `cdest` and `tdest` to make an even number. Meaningless data will be output to the output data position of `num+1`.

The formats of the data values to be input and output using this GBI are defined as follows as the `zNorm`, `zColor`, and `zTxtr` structures in the header file `gZ-Sort.h`.

```
typedef struct      {
  s8                nx, ny, nz;
  }  zNorm;

typedef struct      {
  u8                r, g, b, a;
  } zColor_t;

typedef union       {
  zColor_t          n;
  u32               w;
  }  zColor;

typedef struct      {
  s16               s, t;
  }  zTxtr_t;

typedef union       {
  zTxtr_t           n;
  u32               w;
  }  zTxtr;
```

Since the size of each structure is 3 or 4 bytes, pay special attention to the 8-byte alignment when transferring DMA using `gSPZSetUMem`. When the transfer size must be a multiple of 8, the DMA transfer size must be rounded off to a multiple of 8.

The routine of this GBI is basically as follows. If you are careful not to overwrite the unprocessed `nsrc` and `msrc` with the output of `cdest` and `tdest`, it is possible to overlap these areas.

```
for (i = 0; i < num; i ++) {
     (*cdest, tdest) = CalcLight (*nsrc, *msrc);
     nsrc    += 3;
     msrc    += 4;
     cdest   += 4;
     tdest   += 4;
}
```

`gSPZMixS16` (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor)
`gSPZMixS8` (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor)
`gSPZMixU8` (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor)

`src1`	head address 1 in user area where data to be interpolated is stored and head address (common) in user area from which interpolation results are to be output
`src2`	head address 2 where data to be interpolated is stored
`num`	number of data (multiple of 8)
`factor`	mixed factors (u. 15 format 0x0000~0x7fff value)

These GBIs perform linear interpolation on two numbers using the formula below. The s16, s8, and u8 data types are handled by the respective GBI.

```
(*src1) = (*src1)*factor + (*src2)*(1.0*factor);
```

In `gSPZMixS16`, `src1` and `src2` combined are limited to 16 bytes. Also, in `gSPZMixS8` and `gSPZMixU8`, `src1` and `src2` combined are limited to 8 bytes.

`num` must be a multiple of 8. If a number which is not a multiple of 8 is specified, meaningless data will be output to `src1` until the number becomes a multiple of 8.

Chapter 5 Other Processing

GBI List

This is a list of other GBIs.

`gSPZSegment`	Sets segment
`gSPZSetSubDL`	Registers/starts Sub DL
`gSPZLinkSubDL`	Processes unprocessed Sub DL
`gSPZSendMessage`	Sends message to CPU
`gSPZWaitSignal`	Waits for signal from CPU

GBI Functions

This chapter explains the remaining GBIs.

`gSPZSetSubDL` (Gfx *gp, Gfx *subdl)

`subdl`	`Sub DL head address`

This GBI registers the Sub DL and can only be processed in the Main DL. If a Sub DL has already started, a second Sub DL may not function properly, if entered. Register a Sub DL only after the processing of any Sub DL already registered by `gSPZLinkSubDL` is completed.

`gSPZLinkSubDL` (Gfx *gp)

This GBI processes the Sub DL remaining to be processed and can only be processed in the Main DL. If a Sub DL has already ended, nothing happens when the Sub DL are not registered.

`gSPZSendMessage` (Gfx *gp)

This GBI sends a SP_BREAK message to the CPU to inform the CPU of the status of Display List execution.

When the DL execution status is unknown, the CPU cannot determine whether or not processing has been completed, forcing it to wait until RSP processing has ended (until the RSP message is received).

Display List

| ZObject A vertex calculation
| ZObject B vertex calculation
| ZObject C vertex calculation
↓ At this point, end message is sent to CPU

If the Display List is prepared as shown below using this GBI, the CPU can know whether or not the vertex calculation for each ZObject has ended and can immediately build ZObjects.

Display List

| ZObject A vertex calculation
| gSPZSendMessage → message sent to CPU
| ZObject B vertex calculation
| gSPZSendMessage → message sent to CPU
| ZObject C vertex calculation
↓ gSPZSendMessage → message sent to CPU

Given the overhead resulting from actually sending and receiving messages for each Zobject, as explained above, it may be better to send messages for multiple ZObjects rather than for each object. This is up to the user.

Just as with the delivery of normal messages, for the CPU to receive the `SP_BREAK` message sent from the RSP, a message queue is used. Get the message queue for the `SP_BREAK` message and connect it to `OS_EVENT_SP_BREAK` using `osSetEventMesg`. Also, although it is safer to set the size of the queue to greater than the number of `gSPZSendMessage` in the Display List, this is not necessary. As long as the number of `SP_BREAK` messages can be controlled, a smaller size presents no problem.

In conventional microcode, `rmonThread` used this `SP_BREAK` message. Originally, the message was prepared for microcode `BREAK POINT` processing when using the GameShop DEBUGGER. This function currently is not used significantly, so it was left up to the user. As a result, when `rmonThread` is not used, no problem occurs. When it is used, note that the `SP_BREAK` message queue must be set after creating or starting `rmonThread` (execute `osStartThread` to `rmonThread`).

`gSPZWaitSignal` (Gfx *gp, zSignal *sig, u32 param)

`sig`	pointer to Signal buffer
`param`	Signal value (u32)

This GBI waits until the CPU Signal value exceeds the `param` value. Since the Signal value from the CPU is updated through an RDRAM buffer, that buffer must be contained in the application itself. During execution of this GBI, the RSP determines whether or not the CPU has rewritten the buffer's Signal. If so, the Signal buffer on the RDRAM is DMA transferred to DMEM and compared to the `param`.

The following is a macro for rewriting the CPU's Signal value.

```
GZ_SENSIGNAL (zSignal *sig, u32 val)

sig  pointer to Signal buffer
val  new Signal value
```

After the Signal value is rewritten to `val`, notice that the change that has occurred is sent to the RSP.

Since the Signal value is an unsigned 32-bit variable, the smallest value is 0.

So far in the microcode, the Display List is handed over to the RSP after it is complete. In other words, the RSP cannot process until the Display List has been completely created. However, even if the Display List is not completely created, this GBI can send any created portion to the RSP, i.e., the RSP can be made to wait until the rest of the Display List is created. When this `gSPZWaitSignal` and the earlier output `gSPZSendMessage` are combined, simple synchronicity occurs between CPU and RSP processing, demonstrating the great power of serial processing of the Display List.

Chapter 6 Compatibility With Other Microcodes

About GBIs

Z-Sort Microcode is not compatible with other Fast3D-compatible microcodes. However, some GBIs will be shared to allow switching by the microcode and self-loading of the F3DEX system. This section explains those GBIs that will likely belong to both microcodes.

The names of the GBIs explained here basically have the new prefix `gSPZ` instead of the corresponding prefix `gSP` of the GBI macro in F3DEX.

Z-Sort Microcode GBIs include a subset of F3DEX GBI Level 2. This F3DEX GBI Level 2 is a new and improved GBI set offering faster RSP processing speeds in F3DEX Microcode and will be adopted in the upcoming F3DEX Microcode release.

As a result, Level 2 is not compatible at the binary level with the GBIs adopted in F3DEX Microcode Version 1.23 or earlier. Thus, performing such processing as the microcode and self-loading in the F3DEX microcode system is difficult.

Since Z-Sort Microcode uses F3DEX GBI Level 2, when using Z-Sort Microcode, `F3DEX_GBI_2` must be defined by the `#define` statement or compile option D.

Note: At the present time, `F3DEX_GBI` must also be defined.

Common GBIs

`gSPZSegment`	Sets segment
`gSPZPerspNormalize`	Sets perspective correction value

`gSPZSegment` (Gfx *gp, u32 seg, u32 base)

`seg`	segment number (0~15)
`base`	segment base address

This GBI sets the segment. Although processing by either the Main DL or a Sub DL is possible, when the same segment number has been rewritten in the Main DL or a Sub DL, problems can be expected when parallel processing is started. To avoid these problems, try as much as possible not to overlap the segments to be used in the Main and Sub DL.

`gSPZPerspNormalize` (Gfx *gp, u16 persp)

`persp`	pass correction value

This GBI sets the perspective correction value. It is the same as the `gSPPerspNormalize` GBI in F3DEX.

Chapter 7 CPU Support Library

In Z-Sort Microcode, building plane data from the vertex data on the screen, i.e., ZObject data, depends on the CPU. Using arithmetic operation GBI commands, 3D coordinate vertices can be transformed into screen coordinate vertices. The CPU's role is to connect these vertices to build polygons. The CPU performs other processing as well and, therefore, a CPU library must be created by the user to perform this processing. The library used in the sample program `cubes-1` is explained below to provide a sample library.

- Multiply model matrix by perspective transformation matrix
- Calculate coordinate transformation/perspective transformation/screen depth for model vertices
- Determine whether there are vertices in the screen
- Determine clipping/back plane
- Construct ZObject data
- Create ZObject list

Chapter 8 Sample Programs

The sample programs are installed under the `/usr/src/PR/gZ-Sort` directory.

`zonetri/`

This displays one quadrangle and is the simplest application of Z-Sort.

`cubes-1`

A wide variety of polygons can be drawn using Z-Sort Microcode. The general-purpose library `libZ-Sort` is created and data is sent to it for drawing. Near clipping and other processes are performed in the library. Its 2-pass processing, however, hinders performance.

NINTENDO64

S2DEX Microcode

User's Guide

NUS-06-0136-001A
Released: 1/9/98

D.C.N. NUS-06-0136-001 REV A

"Confidential"

This document contains confidential and proprietary information of Nintendo and is also protected under the copyright laws of the United States and foreign countries. No part of this document may be released, distributed, transmitted or reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from Nintendo.

© 1998 Nintendo

TM® and the "N" logo are trademarks of Nintendo

Table of Contents

Chapter 1. Introduction 5

What is S2DEX Microcode? 5

Features of S2DEX 5
  The Drawing Primitive 5

Self Loading Function 6

DEBUG Information Output Function 6

Passing Commands from RSP to RDP 6

Chapter 2 Compatibility with F3DEX 7

GBIs Supported by Both S2DEX and F3DEX 7

GBIs Not Supported in S2DEX 8

New GBIs 8

Precautions Regarding GBIs 8
  Changing Mode Using OtherMode 8

Chapter 3 S2DEX GBIs 9

BG Drawing GBI 9
  uObjBg Structure 9
  gSPBgRectCopy 11
  gSPBgRect1Cyc 14

The Sprite Drawing GBI 16
  uObjSprite Structure 16
  uObjMtx/uObjSubMtx Structures 17
  gSPObjRectangle 17
  gSPObjRectangleR 20
  gSPObjSprite 20

2D Matrix Operation 21
  gSPObjMatrix 21
  gSPObjSubMatrix 22

Setting the Object Render Mode 22
  gSPObjRenderMode 22
  RenderMode when Drawing Sprites 24

The Texture Load GBI 24
  uObjTxtr Structure 25
  gSPObjLoadTxtr 26

Compound Processing GBI 29
  uObjTxSprite Structure 29
  gSPObjLoadTxRect 29
  gSPObjLoadTxRectR 30
  gSPObjLoadTxSprite 30

Table of Contents (Continued)

Conditional Branching GBI 30
  gSPSetStatus 30
  gSPSelectDL 31
  gSPSelectBranchDL 31

Chapter 4 Emulation Functions 33

guS2DEmuGBgRect1Cyc 33

guS2DEmuSetScissor 33

Chapter 5 DEBUG Information Output Function 35

Chapter 6 Installation of S2DEX Package 37

Chapter 1. Introduction

What is S2DEX Microcode?

The S2DEX microcode has been developed to use Super NES-like sprite and BG functions on the Nintendo 64 (N64). Due to these functions, it is easier to create a game using sprites. Also, by treating drawing objects as sprites and BG, N64 programming is similar to the conventional sprite game programming.

Features of S2DEX

The Drawing Primitive

Since S2DEX is designed specifically for processing 2D expressions, 3D primitive drawing for Fast3D and F3DEX is not supported. However, the following primitives can be drawn using S2DEX Microcode.

Rectangle A -- `gSPObjRectangle`,`gSPObjRectangleR` (Copy Mode)

Size is fixed. Texture flipping (vertical / horizontal) and drawing in the copy mode is possible. Scale change (magnifying / shrinking) and rotation are not possible Texture interpolation display and subpixel movement are not possible. Anti-aliasing processing is not possible. The texture must be loaded to TMEM before drawing.

Rectangle B -- `gSPObjRectangle`, `gSPObjRectangleR`(1, 2 Cycle Mode)

Texture flipping is possible (vertical / horizontal). Drawing in 1, 2 cycle mode is possible. Texture interpolation display and subpixel movement are possible. AntiAlias processing is possible. Scale change (magnifying / shrinking) is possible, but rotation can not be done. Texture must always be loaded to TMEM.

Sprite -- `gSPObjSprite`

Scale change (magnifying / shrinking) and rotation are possible. Texture flipping is possible (vertical / horizontal). Texture interpolation display and subpixel movement are possible. AntiAlias processing is possible. Drawing in copy mode is not possible. Texture must always be loaded to TMEM.

BackGround (BG) A -- `gSPBgRectCopy`

Scrolling in closed region (vertical / horizontal loop) is possible. Horizontal texture flipping is possible (not vertical texture flipping). Drawing is possible in copy mode only. Scale change (magnifying / shrinking) is not possible. Texture interpolation display and subpixel movement are not possible. AntiAlias processing is not possible. Drawing is done by loading the texture on DRAM to TMEM as necessary.

BackGround (BG) B -- `gSPBgRect1Cyc`

A CPU-based emulation routine is available. Scale change (magnifying / shrinking) is possible. Scrolling in closed region (vertical / horizontal loop) is possible. Horizontal texture flipping is possible (not vertical texture flipping). Drawing can be performed in 1 cycle mode only Texture interpolation display is possible. Subpixel movement is possible in the horizontal direction only. AntiAlias processing is not possible. Drawing is done by loading the texture on DRAM to TMEM as necessary.

From Old GBI...

The following functions can be used from the old graphics binary interface (GBI).

- FillRectangle
- TextureRectangle
- TextureRectangleFlip

The following functions can not be used from the old GBI.

- 1Triangle
- 2Triangle
- 1Quadrangle

There are not many similarities between S2DEX and old Sprite2D Microcode. S2DEX is not an upgrade to Sprite2D, but it is rather a new microcode. Also, sprite libraries such as `spInit()` can not be used in combination with S2DEX because sprite libraries use 3D microcode. The S2DEX library is completely different from the sprite library.

Self Loading Function

As mentioned above, S2DEX is not capable of drawing 3D primitives. However, S2DEX has a microcode self loading function which is supported by F3DEX (Release 1.20 or later). Therefore, it is possible for S2DEX to draw 3D primitives by loading F3DEX microcode.

Note: **Use of S2DEX to draw 3D primitives by loading F3DEX microcode requires Release Version 1.22 and later versions of F3DEX Microcode, and N64 OS/Library Version 2.0I or later.**

DEBUG Information Output Function

There are two types of S2DEX Microcode. One is installed for master ROM, and the other is for debugging. The Microcode for debugging is equipped with the following features.

- Output display list processing log
- If illegal input value or illegal command detected, stops RSP, and send the report to CPU

These functions are fully described later in this manual.

Passing Commands from RSP to RDP

S2DEX only supports fifo versions (same as F3DEX series).

However, a larger FIFO buffer is required by S2DEX than for F3DEX. While this buffer had to be 0×300 bytes or larger for the F3DEX series, it has to be at least 0×800 bytes for S2DEX. Please be aware that if you want the FIFO buffer to be shared by the F3DEX series and S2DEX, it must be at least 0×800 bytes to fulfill the S2DEX requirements.

Note: On some of the on-line "Functional Reference Manual" (HTML) pages, the minimum FIFO size in _fifo microcode commands is stated to be 0×100 bytes. This is incorrect. The FIFO size required varies depending on the microcode command. These are as noted above for the F3DEX series and S2DEX, while 0×180 bytes are necessary for Fast3D.

Chapter 2 Compatibility with F3DEX

The GBI of S2DEX was derived from F3DEX. So, there is no compatibility with the GBI of Fast3D. When you use S2DEX, you need to define `F3DEX_GBI` like F3DEX before `ultra64.h` is included.

Also, to use the GBI of S2DEX, you need to include the header file `<PR/gs2dex.h>`. Insert this include specification after the include specification of `<ultra64.h>`.

Next, let's compare the GBI of S2DEX and the GBI of F3DEX. Simply put, you can consider that S2DEX does not support GBIs which deal with 3D primitives, 4x4 matrices, and light definition.

The following refers to `gSP*` and `gDP*` only, but the same applies to `gsSP*` and `gsDP*`.

GBIs Supported by Both S2DEX and F3DEX

The following GBIs are fully supported by both S2DEX and F3DEX, except as noted.

DL Process Control	`gSPDisplayList(*)` `gSPEndDisplayList`	`gSPBranchList`
Setting Up Segment	`gSPSegment(*)`	
Loading Microcode	`gSPLoadUcode*`	
Scissoring	`gDPSetScissor`	`gDPSetScissorFrac`

Setting RDP Mode	`gSPSetOtherMode`
	`gDPSetTexturePersp`
	`gDPSetTextureLOD`
	`gDPSetTextureFilter`
	`gDPSetCombineKey`
	`gDPSetAlphaDither`
	`gDPSetAlphaCompare`
	`gDPSetRenderMode`
	`gDPSetDepthImage`
	`gDPSetCombineMode`
		`gDPSetCycleType`
		`gDPSetTextureDetail`
		`gDPSetTextureLUT`
		`gDPSetTextureConvert`
		`gDPSetColorDither`
		`gDPSetBlendMask`
		`gDPSetDepthSource`
		`gDPSetColorImage`
		`gDPSetTextureImage`

Setting Color Value, etc.	`gDPSetEnvColor`
	`gDPSetFogColor`
	`gDPSetPrimColor`
	`gDPSetConvert`
	`gDPSetKeyGB`
		`gDPSetBlendColor`
		`gDPSetFillColor`
		`gDPSetPrimDepth`
		`gDPSetKeyR`

Loading to TMEM	`gDPSetTileSize`
	`gDPSetTile`
	`gDPLoadMultiBlock*`
	`gDPLoadMultiTile*`
	`gDPLoadTLUT_pal256`
		`gDPLoadTile`
		`gDPLoadTextureBlock*`
		`gDPLoadTextureTile*`
		`gDPLoadTLUT_pal16`

Primitives	`gDPFillRectangle`
	`gSPTextureRectangle`
	`gsSPTextureRectangleFlip`
		`gDPScisFillRectangle`
		`gSPScisTextureRectangle`

Sync Processing	`gDPFullSync` `gDPPipeSync`	`gDPTileSync` `gDPLoadSync`
NOOP	`gSPNoOp` `gDPNoOpTag`	`gDPNoOp`

Note: **For S2DEX, the number of segments when using `gSPSegment` is 16 and the number of DL links when using `gSPDisplayList` is 18 (same as F3DEX).**

GBIs Not Supported in S2DEX

The following GBIs are not supported by S2DEX.

Setting View	`gSPViewport` `gSPPerspNormalize`	`gSPClipRatio`
Matrix Operation	`gSPMatrix` `gSPInsertMatrix`	`gSPPopMatrix` `gSPForceMatrix`
Vertex Operation	`gSPVertex`	`gSPModifyVertex`
Conditional Branch	`gSPCullDisplayList`	`gSPBranchLessZ*`
Polygon Type Setting	`gSPSetGeometryMode` `gSPTexture`	`gSPClearGeometryMode` `gSPTextureL`

Primitives	`gSP1Triangle`
	`gSP1Quadrangle`
	`gSPLineW3D`
		`gSP2Triangles`
		`gSPLine3D`

Lighting	`gSPNumLights`
	`gSPLightColor`
	`gSPLookAt*`
	`gDPSetHilite2Tile`
		`gSPLight`
		`gSPSetLights[0-7]`
		`gDPSetHilite1Tile`

Fog	`gSPFogFactor`	`gSPFogPosition`
For Old Sprite2D Use	`gSPSprite2DBase` `gSPSprite2DDraw`	`gSPSprite2DScaleFlip`

New GBIs

The following GBIs have been added to S2DEX.

BG Drawing	`gSPBgRectCopy`	`gSPBgRect1Cyc`
Sprite Drawing	`gSPObjRectangle` `gSPObjSprite`	`gSPObjRectangleR`
2D Matrix Operation	`gSPObjMatrix`	`gSPObjSubMatrix`
Drawing Mode Setting	`gSPObjRenderMode`	
Load Texture Processing	`gSPObjLoadTxtr`	
Compound Commands	`gSPObjLoadTxRect` `gSPObjLoadTxSprite`	`gSPObjLoadTxRectR`
Conditional Branch	`gSPSelectDL`	`gSPSelectBranchDL`

Precautions Regarding GBIs

Changing Mode Using OtherMode

When changing the mode in F3DEX with `g[s]SPSetOtherMode`, no more than a maximum of 31 bits could be set with a single `g[s]SPSetOtherMode` command. This has been corrected in S2DEX so that you can change 32 bits worth of parameters at once with a single command.

Note: In addition, combining the `g[s]DSPSetOtherMode` command with a `g*SPSetOtherMode`-type command when changing modes resulted in a malfunction with F3DEX. (Normal setting by g*SPSetOtherMode could not be accomplished) This has been corrected in S2DEX so that they can be combined.

Chapter 3 S2DEX GBIs

The following paragraphs contain detailed descriptions of the GBIs available in S2DEX.

BG Drawing GBI

S2DEX can easily create vertical and horizontal scroll surfaces in a closed area (this function was included in the Super NES). Developing scroll games such as 2D Mario will be easier using this feature.

`uObjBg` Structure

`uObjBg` structures hold the drawing information of BG. The pointer to this structure is given as the BG drawing GBI parameter.

`uObjBg` structures can be precisely divided into 3 common structures. The first is for aligning the structure with the 8 byte boundary and does not require attention. The remaining 2 have data structures which adapt for the two BG drawing GBI structures described below.

The structure that adapts for the BG drawing GBI resulting from the Copy Mode is `uObjBg_t` and the structure that adapts for the BG drawing GBI resulting from the 1 Cycle Mode is `uOBjScaleBg_t`.

```
    typedef union {
    uObjBg_t          b;
    uObjScaleBg_t     s;
    long long int     force_structure_alignment;
  } uObjBg;
```

`uObjBg_t` Structure

Members of the `uObjBg_t` structure can be divided into two groups (first half and second half).

The first half consists of the member variables to be set by the user. BG drawing can be controlled by changing these variables. This first half can be shared with the `uObjScaleBg_t` structure.

The second half consists of the variables to be calculated and stored by the CPU to help the Microcode. These member variables are set by calling the function `guS2DInitBg()`, using the `uObjBg` structure's pointer as the parameter. However, there is no need to call `guS2DInitBg` every time.

Since the second half's member variables can be derived from the first half variables (`imageLoad`, `imageFmt`, `imageSiz`, `imageW`, and `frameW`), `guS2DInitBg` needs to be called only immediately after these variables are changed.

Using `uObjBg` as BG plane, these variables don't normally change very often. Therefore, it is usually sufficient to call `guS2DInitBg` once before using BG plane.

However, when the `uObjScaleBg_t` structure's member variables `scaleW`, `scaleH`, `imagYorig` have changed the `uObjBg_t` second half's member variables may be changed. In this situation, it will probably be necessary to call `guS2DInitBg` again.

The following is the definition section of `uObjBg` in `gs2dex.h`. `uObjBg`'s size is 40 bytes; and `uObjBg` must be aligned to 8 bytes.

The first half member variables will be explained in the GBI section. Please understand that the arrangement of member variables is somewhat complicated to optimize RSP operation.

Note: **In S2DEX Version 1.00 and later, the format of member variables `imagePal` and `imageFlip` has changed from u8 to u16.**

```
typedef struct  {
  u16  imageX;       // The x-coordinate of the upper-left position of BG image
(u10.5)
  u16  imageW;       // The width of BG image
(u10.2)
  s16  frameX;       // The upper-left position of the transfer frame
(s10.2)
  u16  frameW;       // The width of the transfer frame
(u10.2)

  u16  imageY;       // The y-coordinate of the upper-left position of BG image
(u10.5)
  u16  imageH;       // The height of BG image
(u10.2)
  s16  frameY;       // The upper-left position of the transfer frame
(s10.2)
  u16  frameH;       // The height of the transfer frame
(u10.2)

  u64  *imagePtr;    // The texture address of the upper-left position of BG image
  u16  imageLoad;    // Which to use, LoadBlock and LoadTile
  u8   imageFmt;     // The format of BG image      G_IM_FMT_*
  u8   imageSiz;     // The size of BG image        G_IM_SIZ_*
  u16  imagePal;     // The pallet number
  u16  imageFlip;    // Image horizontal flip.  Flip using G_BG_FLAG_FLIPS.
  // All of the above are common with uObjScaleBg_t

  // The user doesn't have to set the following since they are set within the
  //initialization routine, guS2DInitBG().
  u16  tmemW;        // The width of TMEM for 1 line's worth of the frame.  The width
                     // is the Word size.
                     // At LoadBlock, GS_PIX2TMEM(imageW/4,imageSiz)
                     // At LoadTile,  GS_PIX2TMEM(frameW/4,imageSiz)+1
  u16  tmemH;        // The width of loadable TMEM at a time. (s13.2)  The height is 4
                           // times value
                     // At the normal texture,    512/tmemW*4
                     // At the CI texture,   256/tmemW*4
  u16  tmemLoadSH;   // The SH value
                     // At LoadBlock, tmemSize/2-1
                     // At LoadTile,   tmemW*16-1
  u16  tmemLoadTH;   // The TH value or the Stride value
                     // At LoadBlock,  GS_CALC_DXT(tmemW)
                     // At LoadTile,   tmemH-1
  u16  tmemSizeW;    // The skip value of imagePtr for 1 line's worth of the image.
                     // At LoadBlock, tmemW*2
                     // At LoadTile,  GS_PIX2TMEM(imageW/4,imageSiz)*2
  u16  tmemSize;     // The skip value of imagePtr for one loading.
                     // = tmemSizeW*tmemH
  } uObjBg_t;        // 40 bytes
```

The following structure defines the initialization function `guS2DInitBg`.

```
Void guS2DInitBg(uObjBg *bg);
```

This function is used for initializing the `uObjBg` structure (`uObjBg_t`). If the `uObjBg` data structure is used as the parameter without initialization, the S2DEX Microcode's GBI may not function properly.

Parameter:	bg	The pointer to the uObjBg structure.

`uObjScaleBg_t` Structure

There is no member argument required to calculate the value by the CPU in advance like `uObjBg_t` in members of the `uObjScaleBg_t` structure. All member arguments are directly set by the user, and BG plane drawing is then controlled accordingly.

In addition, when shared by the `uObjBg` structure, the `uObjScaleBg_t` structure's member variables from `imageX` to `imageFlip` are shared with the `uObjBg_t` structure.

```
typedef struct  {
  u16  imageX;        // The x-coordinate of the upper-left position of BG image
(u10.5)
  u16  imageW;        // The width of BG image
(u10.2)
  s16  frameX;        // The upper-left position of the transfer frame
(s10.2)
  u16  frameW;        // The width of the transfer frame
(u10.2)

  u16  imageY;        // The y-coordinate of the upper-left position of BG image
(u10.5)
  u16  imageH;        // The height of BG image
(u10.2)
  s16  frameY;        // The upper-left position of the transfer frame
(s10.2)
  u16  frameH;        // The height of the transfer frame
(u10.2)

  u64  *imagePtr;    // The texture address of the upper-left position of BG image
  u16  imageLoad;    // Which to use, LoadBlock and LoadTile
  u8   imageFmt;     // The format of BG image     G_IM_FMT_*
  u8   imageSiz;     // The size of BG image       G_IM_SIZ_*
  u16  imagePal;     // The pallet number
  u16  imageFlip;    // Image horizontal flip.  Flipped using G_BG_FLAG_FLIPS.
  // All of the above are common with uObjBg_t

  u16  scaleW;       // The scale value of the x-direction                  (u5.10)
  u16  scaleH;       // The scale value of the y-direction                  (u5.10)
  s32  imageYorig;   // The drawing start-point on image                   (s20.5)

  u8   padding[4];
  } uObjScaleBg_t;   // 40 bytes
```

gSPBgRectCopy

```
gSPBgRectCopy(Gfx *gdl, uObjBg *bg)
gsSPBgRectCopy(uObjBg *bg)
```

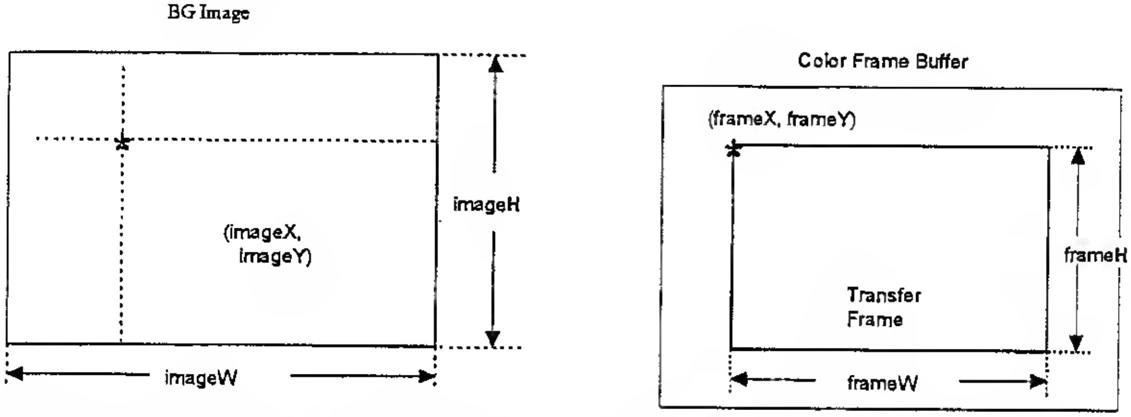
```
Gfx              *gdl;  The display list pointer
uObjBg           *bg;   The pointer to the drawing data structure of BG
```

`g[s]SPBgRectCopy` is the simplest BG drawing GBIs supplied by S2DEX. This GBI has the following features.

- Scale change (magnifying / shrinking) is not possible.
- Scrolling in a closed region (making vertical / horizontal loop) is possible.
- Horizontal texture flipping is possible (not vertical texture flipping).
- Drawing is possible in copy mode only.
- Texture interpolation display and subpixel movement are not possible.
- Anti-aliasing is not possible.
- The GBI loads the texture data from DRAM to TMEM as necessary, then draws.

Designed for drawing in the copy mode, the biggest advantage of `g[s]SPBgRectCopy` is that it has the fastest drawing speed. When using the GBI, CycleType must be set to the Copy mode.

S2DEX sends data from the BG image buffer to the actual frame buffer's rectangle region, shown below. Scrolling becomes possible by establishing the relationship between the upper left hand corner of the frame buffer rectangle region (transfer frame) and a point in the BG image buffer, specified by `imageX` and `imageY`. `imageX`, `imageY` can be specified in the (u10.5) format, but due to restrictions when using the Copy mode, the values for `imageX`, `imageY` are limited to integer values.



The size of the BG image is set by `imageW` and `imageH`. The beginning address (the top left hand corner) is specified by `imagePtr`. That is, you can consider the BG image to be a large texture data having width (`imageW`) and height (`imageH`) starting from `imagePtr`.

BG Image's width, `imageW` must be aligned to 8 bytes. Since the actual values used for `imageW` and `imageH` are in (u10.2) format, the values to be assigned must be multiplied by 4. The following chart shows the `imageW`'s value constraints, taking (u10.2) format into consideration and multiplying by 4. There is no need to align imageH values.

When `G_IM_SIZ_4b` : `imageW` is a multiple of 64

When `G_IM_SIZ_8b` : `imageW` is a multiple of 32

When `G_IM_SIZ_16b` : `imageW` is a multiple of 16

When `G_IM_SIZ_32b` : `imageW` is a multiple of 8

For horizontal scrolling, `imageW` must be larger than `frameW`. The following values take the (u10.2) format into consideration. When `G_IM_SIZ_16b`, `imageW` must be 4 pixels larger than `frameW`.

When `G_IM_SIZ_4b` : `frameW+64 <= imageW`

When `G_IM_SIZ_8b` : `frameW+32 <= imageW`

When `G_IM_SIZ_16b` : `frameW+16 <= imageW`

When `G_IM_SIZ_32b` : `frameW+ 8 <= imageW`

The size of the transfer frame is specified by `frameW` and `frameH`, and the position of the upper left hand corner of the transfer frame on the screen is specified by `frameX` and `frameY`. The parameters of `frameW` and `frameH` are in (u10.2) format. It is possible to specify negative values for `frameX` and `frameY`. If the transfer frame projects out of the scissors box specified by `g[s]DPSetScissor`, the microcode will clip the excess portion.

A problem is not created when the BG frame is bigger than the transfer frame; however, if the transfer frame is bigger than the BG frame, proper operation may not occur. Please be sure to keep a transfer frame smaller than a BG image.

In addition, the right and left ends of BG image are offset in the Y direction by 1. Specifically, a BG image's right end pixel is at (`imageW`-1, n), and one pixel to the right is (0, n+1). This arrangement is necessary to improve RDRAM access efficiency for loading texture. It is very important for application developers to keep this in mind.

Texture format and size for a BG image will be set by specifying `imageFmt` and `imageSiz` using the macros: `G_IM_FMT_*`, and `G_IM_SIZ_*`, respectively. Also, when using CI4 texture, assign TLUT number to `imagePal`.

There are two ways to load texture for a BG image--using `LoadBlock` and using `LoadTile`. Since there are advantages and disadvantages for each method, S2DEX's GBI design allows the user to select the proper method by setting a member variable (`imageLoad`). Depending on the situation, the user can assign an appropriate value (`G_BGLT_*`) to `imageLoad` to use `LoadBlock` or `LoadTile`.

The value of `imageLoad`	Meanings
`G_BGLT_LOADBLOCK`	Use `LoadBlock`
`G_BGLT_LOADTILE`	Use `LoadTile`

When using `LoadBlock`, maximum performance can be gained under certain circumstances. However, when certain conditions are not satisfied, `LoadBlock` can not be used because processing overhead will become too large. On the other hand, `LoadTile` can always perform at a certain level. We recommend using `LoadBlock` when the maximum benefit is expected, and use `LoadTile` in other cases.

`LoadBlock`'s use is limited by the width of BG. When `imageSiz` is 16 bit, the possible values of `imageW` usable for `LoadBlock` are the following:

4, 8, 12, 16, 20, 24, 28, 32, 36, 40,
48, 64, 72, 76,100,108,128,144,152,164,
200,216,228,256,304,328,432,456,512,684,
820,912

When `imageSiz` is 8 bit long, the usable set of numbers for `imageW` can be obtained by doubling each of the numbers above. Similarly, multiply each number by 4 when `imageSiz` is 4 bit, and multiply each number by 1/2 when `imageSiz` is 32 bit. This is consistent with the chart in the N64 Programming Manual, Chapter 12, Appendix A, "LoadBlock Line Limits". If the width of the BG image does not allow the use of `LoadBlock`, `LoadTile` must be used.

In order to draw a transfer frame line by line, `LoadBlock` reads the entire line of the corresponding BG image. Since scrolling BG requires a larger BG image for BG refresh, `imageW` must be greater than `frameW`. For this reason, excess data will be loaded when using `LoadBlock`.

On the other hand, `LoadTile` loads necessary data only. Since the processing speed of `LoadBlock` is faster than that of `LoadTile`, using `LoadBlock` is advantageous when the difference of loaded data is only a few pixels. However, when `imageW` is much larger than `frameW`, the processing overhead could become too high. The use of `LoadTile` is advantageous in this case. The user should choose the command best suited for the given application.

As an example, let's assume we are using BG to cover the entire screen (320 X 240).

Since the transfer frame is the entire screen, `frameW` becomes 320 pixels. Reserving 8 pixels for the BG refresh area, `imageW` is 328 pixels. In this case, the difference between `frameW` and `imageW` is small; and using `LoadBlock` at 328 pixels is the best solution.

The GBI supports BG image flipping for the horizontal direction only. A texture image can be flipped by assigning `G_BG_FLAG_FLIPS` to `imageFlip`. Assign 0 for normal display (no flipping).

`gSPBgRect1Cyc`

```
gSPBgRect1Cyc(Gfx *gdl, uObjBg *bg)
gsSPBgRect1Cyc(uObjBg *bg)

Gfx              *gdl;  The display list pointer
uObjBg           *bg;   The pointer to the BG drawing data structure
```

`g[s]SPBgRect1Cyc` is one of the BG drawing GBIs provided by S2DEX, whereby the BG screen can be enlarged or reduced. The features of this GBI are listed below.

- Scale change (magnifying / shrinking) is possible.
- Scrolling in a closed region (making vertical / horizontal loop) is possible.
- Horizontal texture flipping is possible (not vertical texture flipping).
- Drawing in 1 Cycle mode only.
- Texture interpolation display is possible, subpixel movement is possible only in the horizontal direction.
- Anti-aliasing is not possible.
- The GBI loads the texture data from DRAM to TMEM, then draws.

Note: This GBI can not be used in Copy mode.

The parameters necessary for drawing with `g[s]SPBgRect1Cyc` are the parameters required when using `g[s]SPBgRectCopy`, discussed previously, plus the parameters `scaleW`, `scaleH`, and `imageYorig`. The additional parameters will be explained here.

The biggest difference between `g[s]SPBgRect1Cyc` and `g[s]SPBgRectCopy` is that it supports BG scaling. BG scaling is controlled by the `uObjScaleBg_t` structure's member variables `scaleW` and `scaleH`. This scaling is centered at the BG image's (`imageX`, `imageY`).

In other words, even when scaling has been performed, BG image's (`imageX`, `imageY`) are drawn at the position of (`frameX`, `frameY`) in the frame buffer, just as if scaling had not been done. (However, if horizontal flipping has been performed, they are drawn at the position, (`frameX+frameW-1, frameY`).

In addition, when magnifying, the image is clipped by the frame size. Conversely, when shrinking, the frame is sometimes clipped by image size. Refer to the S2DEX sample program for more about this.

However, frame clipping during shrinking can sometimes be slightly greater or lesser depending on calculation error. When a precise size is required, calculate and set the values for `frameW` and `frameH` on the CPU side.

Bilinear interpolation display is supported by `g[s]SPBgRect1Cyc`. When using bilinear interpolation display, jagged lines in texels become less apparent in magnification compared with normal point sampling display, giving a smoother appearance. However, this effect is less apparent in images which are scaled down in size.

When bilinear interpolation is used, the RDP drawing performance decreases compared to when it is not used. The rate of this decrease in performance is greater when a smaller number of image lines are loaded in TMEM at one time. When drawing a 320X240 image in a 320X240 frame with no scaling is compared to drawing a 640x480 image at 1/2 reduction, the share of overhead taken by using bilinear interpolation will be greater when shrinking the 640x480 image. This causes a substantial drop in performance when a 640x480 image is similarly reduced and displayed using point sampling. Considering that the effects of bilinear interpolation diminish when used in reducing images, as discussed above, you should probably consider switching to point sampling display when reducing an image.

`g[s]SPBgRect1Cyc` draws an image by automatically dividing it into several subplanes, but it is possible that the drawing result will unexpectedly develop unnatural wrinkles during the division process if the division is done carelessly. This is especially noticeable when the image is scrolled. The member variable `imageYorig` has been provided for `uObjScaleBg_t` to prevent these wrinkles. The value of `imageYorig` refers to the Y coordinate of the origin for scaling, but it also describes the division origin of a subplane. It is thus possible to prevent the wrinkles described above. Typically, `imageYorig` is used in the following situations.

At initialization:

> Set the value of `imageY` to `imageYorig`.

When the value of `scaleH` changes:

> Set the value of `imageY` to `imageYorig`.

When `imageX` and `imageY` have been wrap processed:

> Perform the same processing that was performed in `imageY` on `imageYorig`.

When changing only `imageY` (change not accompanying wrap processing):

> Do not change `imageYorig`.

Based on the above, processing for an image which is being scrolled by `dx` and `dy` would be as follows.

```
/*Addition of scroll values. */
bg->s.imageX += dx;
bg->s.imageY += dy;

/* Wrap processing of the screen edge. */
if (bg->s.imageX < 0){
  bg->s.imageX += bg->s.imageW;
  bg->s.imageY      -= 32;
  bg->s.imageYorig -= 32;
}
if (bg->s.imageX >= bg->s.imageW){
  bg->s.imageX      -= bg->s.imageW;
  bg->s.imageY      += 32;
  bg->s.imageYorig += 32;
}
if (bg->s.imageY < 0){
  bg->s.imageY      += bg->s.imageH;
  bg->s.imageYorig += bg->s.imageH;
}
if (bg->s.imageY >= bg->s.imageH){
  bg->s.imageY      -= bg->s.imageH;
  bg->s.imageYorig -= bg->s.imageH;
}
```

BG images can be flipped in the horizontal direction only with this GBI and functions just like it does in the COPY mode. The texture image can be flipped by substituting `G_BG_FLAG_FLIPS` for the member variable `imageFlip`. For normal display (no flipping) substitute 0.

When using this GBI, there are limitations on the value of the `uObjScaleBg_t` structure's member variable, `imagePtr`. Any position from the head of RDRAM to the 4096 byte position cannot be specified as the value for `imagePtr`. This represents physical addresses 0x00000000 to 0x00000fff, in which range `imagePtr` (after segment conversion) cannot be placed. Please keep this in mind.

This GBI is built into S2DEX Version 1.00 and later.

In addition, the function `guS2DEmuBgRect1Cyc` has been added, beginning with S2DEX Version 0.75. This function emulates processing which is equivalent to `gSPBgRect1Cyc` by combining several GBIs, such as `gSPTextureRectangle`, etc. This can also be used for performing scaleable BG drawing. See Chapter 4 for details.

The Sprite Drawing GBI

The sprite mentioned here corresponds to OBJECTs in Super NES programming. Sprites have been used for drawing areas smaller than BG, and historically they have been used as "player characters" quite often. In S2DEX, magnifying / reducing, and rotation of sprites are all possible. Also using sprites, more natural expression is possible due to the use of bilinear interpolation processing.

To support a sprite's rotation, a two dimensional coordinate conversion matrix is used. By setting the matrix's elements, a sprite can be rotated freely. The matrix must be set before drawing a sprite. Also, unlike the matrix for Fast3D or F3DEX, there is no matrix stack; so Push/Pop operation can not be performed. Matrix multiplication can not be done either. Only the load operation is possible. (Please refer to, "2D Matrix Operation" on page 21.)

S2DEX specifications call for using separate GBIs for TMEM loading and sprite drawing. In other words, before drawing a sprite, the texture used for the sprite must already be loaded using the texture load GBI (Please refer to, "Texture Load GBI" on page 24.).

The sprite drawing mode can be divided into two categories, rotating sprites and non-rotating sprites. For each respective case, the corresponding GBI will do the processing.

The Drawing Mode	Corresponding GBI
No Rotation	`g[s]SPObjRectangle`, `g[s]SPObjRectangleR`
Rotation	`g[s]SPObjSprite`

`uObjSprite` Structure

The `uObjSprite` data structure holds a sprite's information. The pointer to the data structure will be given to the sprite drawing GBI as a parameter.

```
typedef struct {
  s16  objX;          // The x-coordinate of the upper-left end of OBJ. (s10.2)
  u16  scaleW;        // The width of direction scaling. (u5.10)
  u16  imageW;        // The width of the texture. (The length of the S
                      //.direction.) (u10.5)
  u16   paddingX;     // Unused.  Always 0.

  s16  objY;          // The y-coordinate of the upper-left end of OBJ. (s10.2)
  u16  scaleH;        // Scaling of the height direction. (u5.10)
  u16  imageH;        // The height of the texture. (The length of the T
                      // direction.) (u10.5)
  u16  paddingY;      // Unused.  Always 0.

  u16  imageStride;   // The folding width of the texel.  (In units of 64-bit word.)
  u16  imageAdrs;     // The texture starting position in TMEM. (In units of 64-bit
                      // word.)

  u8   imageFmt;      // The format of the texel.  G_IM_FMT_*
  u8   imageSiz;      // The size of the texel.       G_IM_SIZ_*
  u8   imagePal;      // The pallet number.
  u8   imageFlags;    // The display flag.

 } uObjSprite_t;      // 24 bytes

 typedef union {
 uObjSprite_t        s;
 long long int       force_structure_alignment;
 } uObjSprite;
```

Although the sequence of member variables is somewhat complicated, this is unavoidable to optimize RSP processing (same as with `uObjBg`).

`uObjMtx/uObjSubMtx` Structures

S2DEX Microcode has the area to hold a 2D matrix for controlling a Sprite's rotation. There are eight parameters (`A, B, C, D, X, Y, BaseScaleX`, and `BaseScaleY`).

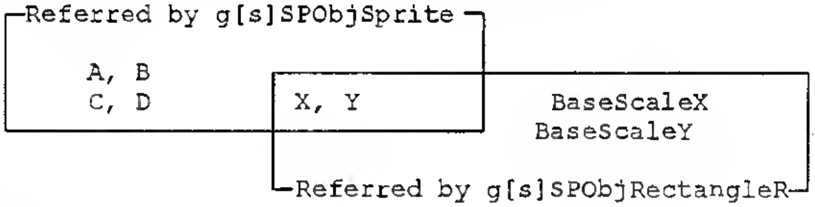
`uObjMtx` data structure has one-to-one correspondence to this 2D matrix area, and the structure is used for modifying the whole 2D matrix. Rotation operation using the 2D matrix is explained in "`gSPObjSprite`" on page 20.

```
typedef struct {
  s32  A, B, C, D;   /* s15.16 */
  s16  X, Y;         /* s10.2  */
  u16  BaseScaleX;   /* u5.10  */
  u16  BaseScaleY;   /* u5.10  */
  } uObjMtx_t;       /* 24 bytes */

  typedef union {
  uObjMtx_t  m;
  long long int force_structure_alignment;
  } uObjMtx;
```

`uObjSubMtx` is a subset of `uObjMtx`, and is used for changing `X, Y, BaseScaleX`, and `BaseScaleY`. The main use for `uObjSubMtx` is drawing a sprite using `g[s]SPObjRectangleR`. Please refer to "`gSPObjRectangleR`" on page 19 for details.

```
typedef struct {
  s16  X, Y;         /* s10.2  */
  u16  BaseScaleX;   /* u5.10  */
  u16  BaseScaleY;   /* u5.10  */
  } uObjSubMtx_t;    /* 8 bytes */

  typedef union {
  uObjSubMtx_t  m;
  long long int force_structure_alignment;
  } uObjSubMtx;
```

The eight elements of a 2D matrix (`A, B, C, D, X,Y, BaseScaleX`, and `BaseScaleY`) can be referenced by `g[s]SPObjSprite` and `g[s]SPRectangleR`. However, not all 8 elements are actually referenced (please refer to the chart below). `X`, and `Y` are referenced by both.

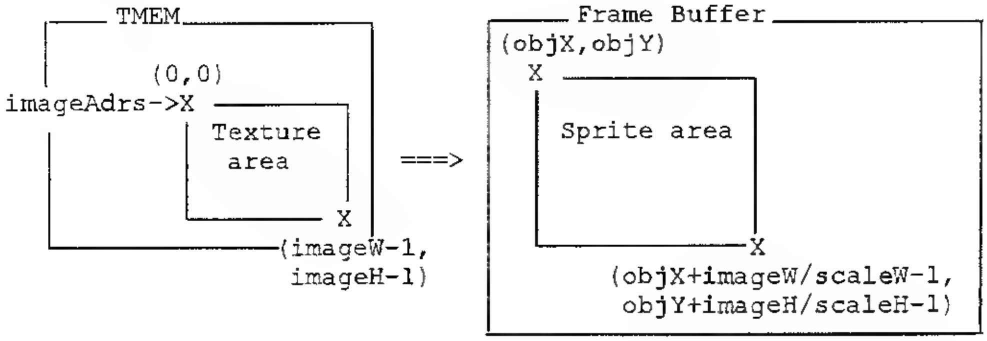


`gSPObjRectangle`

```
gSPObjRectangle(Gfx *gdl, uObjSprite *sp)
gsSPObjRectangle(uObjSprite *sp)

Gfx              *gdl;  The display list pointer.
uObjSprite       *sp;   The pointer to the structure of the sprite drawing data.
```

`g[s]SPObjRectangle` is one of the sprite drawing GBIs supplied by S2DEX and used for non-rotating sprite drawing. The process inside the RSP is to create the `TextureRectangle` command from the input `uObjSprite` structure data and send it to the RDP.

The `g[s]SPObjRectangle` GBI draws texture for the rectangle area defined by the upper left hand corner screen coordinate (`objX, objY`), and lower right hand corner screen coordinate (`objX+imageW/scaleW-1, objY+imageH/scaleH-1`). The drawn texture region will be defined by upper left hand corner (`0,0`) and lower right hand corner (`imageW-1, imageH-1`). If `scaleW` and `scaleH` are 1 << 10, texture will be drawn with equal proportions, without scaling. Please refer to the following page.



Also, when a sprite is drawn, the scissors box defined by `gDPSetScissor` is referenced, and automatic drawing area clipping is done. Therefore, it is possible to set negative values for `objX` and `objY`.

The TMEM address corresponding to the origin of texture region (0,0) can be specified by `imageAdrs`. Normally, `imageAdrs` is set as the beginning of the TMEM loading location specified by the texture load GBI. It is convenient to use the `GS_PIX2TMEM()` macro for this operation. `GS_PIX2TMEM()`, which is defined in `gs2dex.h`, is the macro used to convert a pixel unit number to a TMEM address number.

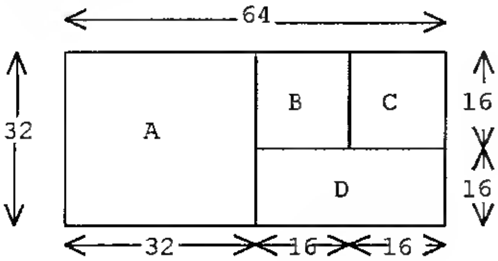
- GS_PIX2TMEM(pix,siz)
- pix: The number of pixels
- siz: The size of 1 texel. Specified by G_IM_SIZ_*

The horizontal width (folding width) at the time of texture load is assigned to `imageStride`. The reason for this is that sometimes the loaded texture width and the `imageW` of the actual sprite drawn are different. Since this is also specified in the TMEM address unit, `GS_PIX2TMEM()` can be used.

An application using `imageAdrs` and `imageStride` is introduced, as follows. Load the multiple of small texture (subtexture) in TMEM first. Now the user can choose the appropriate drawing texture by setting the `imageAdrs` as shown below.

```
imageW      = (sub-texture width);
imageH      = (sub-texture height);
imageAdrs   = GS_PIX2TMEM((S-coordinate in TMEM)+(T-coordinate in TMEM)*
                              (texture width at load time),G_IM_SIZ_*);
imageStride = GS_PIX2TMEM(texture width at load time);
```

More specifically, prepare a large texture consisting of 4 textures, as follows:



Load this composite texture as a 64 x 32 texture; and when drawing a sprite, specify each texture as follows:

```
Sub-texture A:imageW = 32;
              imageH = 32;
              imageAdrs   = GS_PIX2TMEM(0*64+0,   G_IM_SIZ_16b);
              imageStride = GS_PIX2TMEM(64, G_IM_SIZ_16b);

Sub-texture B:      imageW = 16;
              imageH = 16;
              imageAdrs   = GS_PIX2TMEM(0*64+32,  G_IM_SIZ_16b);
              imageStride = GS_PIX2TMEM(64, G_IM_SIZ_16b);

Sub-texture C:      imageW = 16;
              imageH = 16;
              imageAdrs   = GS_PIX2TMEM(0*64+48,  G_IM_SIZ_16b);
              imageStride = GS_PIX2TMEM(64, G_IM_SIZ_16b);

Sub-texture D:      imageW = 32;
              imageH = 16;
              imageAdrs   = GS_PIX2TMEM(16*64+32,  G_IM_SIZ_16b);
              imageStride = GS_PIX2TMEM(64, G_IM_SIZ_16b);
```

There is a limitation to this method however. The format for storing data at TMEM is different for an odd numbered line and an even numbered line. In the calculation formula for `imageAdrs` (T coordinate in TMEM), you can not specify an odd number value.

When using `g[s]SPObjRectangle`, the format and size of the texture is specified by setting `imageFmt` and `imageSiz` using the macros `G_IM_FMT_*`, and `G_IM_SIZ_*`. Also, if CI4 texture is used, specify `imagePal` using TLUT number.

`g[s]SPObjRectangle` supports texture pattern flipping in the `S` and `T` directions. The drawing direction can be changed by setting the following values.

Value of `imageFlags`	Drawing Effect
`0`	`No flipping`
`G_OBJ_FLAG_FLIPS`	`The inversion of the S direction (X)`
`G_OBJ_FLAG_FLIPT`	`The inversion of the T direction (Y)`
`G_OBJ_FLAG_FLIPS|G_OBJ_FLAG_FLIPT`	`The inversion of the S (X) and T (Y) directions`

`g[s]SPObjRectangle` can be used for 1 cycle, 2 cycle, and copy modes. Drawing speed using copy mode is faster than other modes; however, there are more drawing restrictions using copy mode.

Copy mode does not support bilinear interpolation, subpixel processing, and enlarging/reducing in the X direction. If these operations are attempted in copy mode, they may not be performed properly. In the worst case, the RDP may become uncontrollable. We recommend selecting the proper mode to perform necessary functions.

The drawing result using `g[s]SPObjRectangle` will vary depending on the render mode, such as; bilinear interpolation, etc. Please refer to, "Setting the Object Render Mode" on page 22 for details.

`g[s]SPObjRectangle` does not reference the 2D matrix setting. For this reason, the 2D matrix setting does not affect this GBI's drawing result.

gSPObjRectangleR

```
gSPObjRectangleR(Gfx *gdl, uObjSprite *sp)
gsSPObjRectangleR(uObjSprite *sp)
```

```
Gfx              *gdl;  The display list pointer
uObjSprite       *sp;   The pointer to the structure of the sprite drawing data
```

`g[s]SPObjRectangleR` is one of the sprite drawing GBIs provided by S2DEX. Like `g[s]SPObjRectangle`, `g[s]SPObjRectangleR` is used for drawing non-rotating Sprites. Unlike `g[s]SPObjRectangle` however, `g[s]SPObjRectangleR` changes drawing screen coordinates by referring to the 2 D matrix.

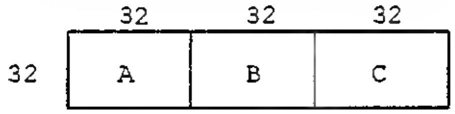
`G[s]SPObjRectangleR` refers to `X, Y, BaseScaleX,` and `BaseScaleY` in the 2D matrix, and determines the vertex coordinates of a sprite using the following formula.

```
Upper-left hand coordinate  ( X + objX / BaseScaleX, Y+objY/BaseScaleY )
Lower-right hand coordinate ( X + (objX + imageW / scaleW) / BaseScaleX - 1,
                              Y + (objY + imageH / scaleH) / BaseScaleY - 1 )
```

To change the values in {`X, Y, BaseScaleX, BaseScaleY`}, use the `g[s]SPObjSubMatrix` GBI. When `X = Y = 0` and `BaseScaleX = BaseScaleY = 1.0`, the result is the same as using `g[s]SPObjRectangle`.

By changing the values in {`X, Y, BaseScale X, BaseScale Y`} of the 2D matrix, multiple Sprites can be moved or their scale changed, as if they were one sprite.

For example, consider the arrangement of the three Sprites A, B, and C in the following example:



and set the (`objX, objY`) data as follows.

```
A: (objX, objY) = ( 0<<2, 0<<2)
B: (objX, objY) = (32<<2, 0<<2)
C: (objX, objY) = (64<<2, 0<<2)
```

Now, by changing X and Y in this example, the three Sprites will move as one sprite.

However, because of a calculation error (performing multiplication for example) sometimes gaps are created between A and B or between B and C. To solve this problem, the adjacent Sprites are slightly overlapped (see below).

```
B: (objX, objY) = ((32<<2)-2, 0<<2)
C: (objX, objY) = ((64<<2)-4, 0<<2)
```

This completes the explanation of the differences between `g[s]SPObjRectangleR` and `g[s]SPObjRectangle`. For other features of `g[s]SPObjRectangleR`, please refer to "`g[s]SPObjRectangle`" on page 17.

gSPObjSprite

```
gSPObjSprite(Gfx *gdl, uObjSprite *sp)
gsSPObjSprite(uObjSprite *sp)
```

```
Gfx              *gdl;  The display list pointer
uObjSprite       *sp;   The pointer to the structure of the sprite drawing data
```

`g[s]SPObjSprite` is one of the sprite drawing GBIs provided by S2DEX. This GBI is used for drawing rotating sprites. To rotate a sprite, use {`A, B, C, D, X, Y`} of the 2D matrix. `g[s]SPObjMatrix` is used for setting these elements of the 2D matrix. (Please refer to "`gSPObjMatrix`" on page 21.)

A point (`x, y`) on a non-rotating sprite will move to the point (`x', y'`) by performing 2D matrix multiplication as follows.

```
x'=A * x + B * y + X
y'=C * x + D * y + Y
```

Each vertex of the sprite will move, and the sprite is drawn in the new region defined by the new vertices.

If the 2D matrix (`A, B, C, D`}is defined by the rotation matrix as follows, a sprite will make a T rotation.

$$\begin{vmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{vmatrix} = \begin{vmatrix} \mathbf{cosT} & \mathbf{sinT} \\ -\mathbf{sinT} & \mathbf{cosT} \end{vmatrix}$$

In this case, a sprite will rotate centering around the screen coordinate (`X,Y`). If scaling is to be added, multiply each element {`A, B, C, D`} by the scale value.

By changing (`objX, objY`), the rotation center of a sprite (`X,Y`) can be changed. If `objX=objY=0`, a Sprite's rotation center will be the upper left hand vertex. If you wish to rotate a sprite about its center, set `objX`, and `objY` as follows.

```
ObjX = -(imageW/scaleW)/2;
objY = -(imageH/scaleH)/2;
```

Also, similar to `g[s]SPObjRectangleR`, by adjusting the values of `objX` and `objY`, multiple Sprites can be rotated as if they were one sprite. Here, as with `g[s]SPObjRectangleR`, we recommend drawing Sprites in a slightly overlapping fashion to eliminate gaps caused by calculation errors.

By setting (`A = D = 1.0, B = C = 0.0`), a non-rotating sprite's location will coincide with a sprite drawn with `g[s]SPObjRectangleR` by setting `BaseScaleX = BaseScaleY = 1.0`. We recommend drawing a non-rotating sprite with `g[s]SPObjRectangle`, and using `g[s]SPObjSprite` for rotating Sprites. Since `g[s]SPObjSprite` uses two polygons in combination for drawing, it requires more RSP/RDP processing than using `g[s]SPObjRectangleR`.

Also, when using `g[s]SPObjSprite` for a non-rotating sprite, a magnified sprite drawing may not coincide with the drawing done by `g[s]SPObjRectangle`. This is unavoidable since the drawing methods are different (polygon combination vs. rectangle drawing).

The setting for the texture to be placed on a sprite is the same as `g[s]SPObjRectangle`. Please refer to the appropriate section above.

2D Matrix Operation

As mentioned above, S2DEX Microcode uses a 2D matrix as the drawing parameter. Several GBIs are provided for the purpose of modifying this 2D matrix.

`gSPObjMatrix`

```
gSPObjMatrix(Gfx *gdl, uObjMtx *mtx)
gsSPObjMatrix(uObjMtx *mtx)

Gfx             *gdl;  The display list pointer
uObjMtx         *mtx;  The pointer to the 2D matrix structure
```

Load the 2D matrix parameter in the `uObjMtx` structure to the 2D matrix area in the RSP. Usually, this GBI is used for a rotating sprite.

Since only 6 matrix elements (`A, B, C, D, X, Y`) are needed for rotation processing, it appears that there is no need to transfer the entire 2D matrix. However, 24 bytes including {`BaseScaleX`, `BaseScaleY}` are transferred, because an 8 byte unit must be maintained for transfer from main memory to the RSP matrix region.

For this reason, the values of `BaseScaleX` and `BaseScaleY` are always overwritten. If you are not using these parameters (not using `g[s]SPObjRectangleR` immediately after calling gSPObjMatrix), we recommend assigning the default value of 1024 (1.0 for s5.10 format) to `BaseScaleX` and `BaseScaleY`.

gSPObjSubMatrix

```
gSPObjSubMatrix(Gfx *gdl, uObjMtx *mtx)
gsSPObjSubMatrix(uObjMtx *mtx)

Gfx            *gdl;  The display list pointer
uObjSubMtx     *mtx;  The pointer to the 2D matrix structure
```

`g[s]SPObjSubMatrix` loads the data in the `uObjSubMtx` structure to the 2D matrix region of the RSP. However, the `uObjSubMtx` structure is a subset of `uObjMtx`, and holds the values of 2D matrix elements {`X, Y, BaseScaleX, BaseScaleY`} used by `g[s]SPObjRectangleR`.

This GBI changes 2D matrix elements {`X, Y, BaseScaleX, BaseScaleY`} corresponding to the variable of uObjSubMtx structure only, and it does not affect the values in {`A, B, C, D`}.

This GBI is used mainly in conjunction with `g[s]SPObjRectangleR`.

Setting the Object Render Mode

Many drawing parameters exist in the RDP, which control sprite/BG drawing. Depending on the RDP mode, polygon drawing and rectangle drawing processes are affected in some subtle ways. For example, by setting bilinear interpolation on and off, texture coordinates will vary by 0.5. S2DEX Microcode has been designed to correct these effects at the RSP to minimize the user's efforts to get around these problems. The RSP's correction process corresponds to the RDP's mode. We call the RSP's correction mode "Object render mode" (or OBJ render mode).

Automatic selection of this mode will increase the processing overhead of the RSP; so currently Copy Mode and 1,2CycleMode have the benefit of automatic operation. For other modes, it is necessary to let the RSP know in the form of the GBI. The current Object render mode has an independent rendering function, in addition to the capability to correct the effects caused by changing the RDP's mode. See the next paragraph for the details.

gSPObjRenderMode

```
gSPObjRenderMode(Gfx *gdl, u32 mode)
gsSPObjRenderMode(u32 mode)

Gfx            *gdl;  The display list pointer
u32            mode;  The Object render mode
```

`g[s]SPObjRenderMode` is used for changing the Object render mode of the RSP. Usually, Object render mode is set based on the display mode.

The flags used are shown below. If multiple settings are required, connect the conditions using the OR operator. However, `G_OBJRM_SHRINKSIZE_1` and `G_OBJRM_SHRINKSIZE_2` can not be used at the same time.

Macro Name	Function
`G_OBJRM_NOTXCLAMP`	does not perform clamp operation for peripheral part of the texture
`G_OBJRM_BILERP`	switches to on for bilinear interpolation
`G_OBJRM_SHRINKSIZE_1`	cut 0.5 texel around the image
`G_OBJRM_SHRINKSIZE_2`	cut 1.0 texel around the image
`G_OBJRM_WIDEN`	expand the image by 3/8 texel

Each flag is explained in detail below:

G_OBJRM_NOTXCLAMP

To place texture on a sprite, the following relationships exist among texture size (`imageW` and `imageH`), scale values (`scaleW` and `scaleH`), sprite size, (`objW` and `objH`).

```
objW = imageW / scaleW;
objH = imageH / scaleH
```

When placing texture on the sprite, the region `(0,0)-(imageW-1, imageH-1)` in the texture coordinates will be displayed on the sprite. However, sometimes texture slightly outside of this region may be displayed, exceeding the outermost edge of the sprite.

To prevent this from occurring, the RSP performs a clamping operation for the excess texture outside of the defined region. For details on this clamping operation, please refer to Chapter 12 of the N64 Programming Manual, "Texture Mapping".

The flag `G_OBJRM_NOTXCLAMP` causes the RSP not to perform this clamping operation. Normally it is not necessary to set this flag to "ON".

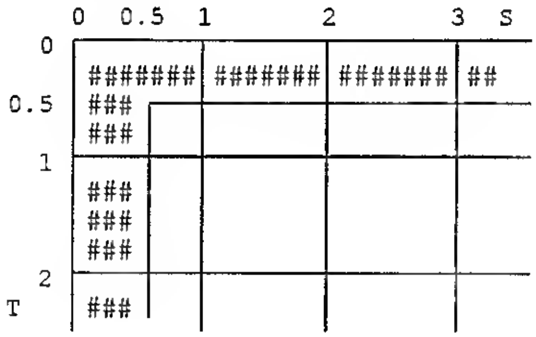
G_OBJRM_BILERP

This flag is set when using texture bilinear interpolation. As we have explained above, the texture discrepancy of 0.5 due to bilinear interpolation will be corrected by setting this flag.

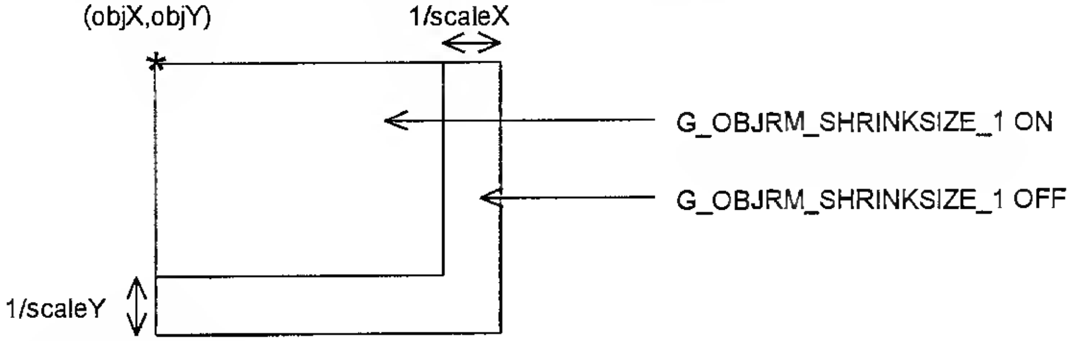
Also, when this flag is ON, the RSP supports internal image movement by subpixel units, using bilinear interpolation. As a result, a sprite can be moved by 1/4 pixel units.

G_OBJRM_SHRINKSIZE_1

When combining multiple bilinear interpolated Sprites and treating them as one large bilinear interpolated sprite, care must be taken to assure continuity of the images at boundary lines. To maintain the continuity between the images, it is necessary to overlap each Sprite's texture by one line. If this is done, 0.5 texel (denoted by # in the chart below) from outer edge will become unnecessary, since this portion will be covered by the adjacent sprite.



When the flag `G_OBJRM_SHRINKSIZE_1` is ON, the RSP will shrink the Sprite's drawing image by eliminating 0.5 texel, and draw the texture image. The texture image will shrink by 0.5, but the upper left hand corner coordinate will not change. The resultant drawing becomes:



G_OBJRM_SHRINKSIZE_2

This is similar to `G_OBJRM_SHRINKSIZE_1`. The only difference is that the amount of image shrinkage is doubled (1 texel from the outer edge).

This flag is used for overlapping adjacent Sprites' texels by two lines for better continuity for subpixel processing.

G_OBJRM_WIDEN

This expands the image by 3/8 texel in the positive `S`, `T` directions.

This flag is used to prevent blank spaces from opening at the seams when Sprites are combined to display a rotating Object which is larger than TMEM.

The importance of this flag was decreased as calculation for sprite rendering is processed more precisely with S2DEX Version 1.04 and later, however this flag is still usable.

Note: The following flags are no longer supported at the time of this manual's release.

- `G_OBJRM_ANTIALIAS`
- `G_OBJRM_XLU`

RenderMode when Drawing Sprites

The RenderMode of the RDP which needs to be set for rendering a sprite is defined in a header file, `gs2dex.h`. Please use this when rendering a sprite.

For Anti-aliasing off:

Opaque sprite	`G_RM_SPRITE*`
Semi-transparent sprite	`G_RM_XLU_SPRITE*`

For -aliasing on:

Opaque sprite	`G_RM_AA_SPRITE*` (`G_RM_RA_SPRITE*`)
Semi-transparent sprite	`G_RM_AA_XLU_SPRITE*`

When a semi-transparent sprite is used for Anti-aliasing On, and two sprites are layered, sometimes the edge portion of the sprite which is layered on the bottom may affect the edge portion of the sprite on top. Since this is inevitable, please use `G_RM_XLU_SPRITE` if this is unacceptable.

The Texture Load GBI

The sprite drawing process for S2DEX was described in the sprite GBI section. Here, we will describe the TMEM load process, which is another important operation.

`uObjTxtr` Structure

In the Texture Load GBI, three different texture types are processed by the same GBI. These three different types (methods) are distinguished by `uObjTxtr` structure's member variable `type`, which is provided to the GBI. These three methods are shown below.

1. Texture load using LoadBlock
2. Texture load using LoadTile
3. TLUT load

Texture load using `LoadBlock` can be faster than texture load using `LoadTile`; however, there is a limitation to loadable texture width. Since this limitation is the same as "LoadBlock"; please refer to page 13 for details.

Corresponding to the three different methods, three different data structures are defined. These data structures are constructed the same way, having different member variable names. These data structures are combined into a union (`uObjTxtr` structure).

1. Texture load structure `uObjTxtrBlock_t` for using `Loadblock`

```
typedef      struct {
  u32  type;          // by type G_OBJLT_TXTRBLOCK
  u64  *image;        // texture source address on DRAM
  u16  tmem;          // TMEM word address of loading destination (8byteWORD)
  u16  tsize;         // texture size specified by macro GS_TB_TSIZE()
  u16  tline;         // texture width specified by macro GS_TB_TLINE()
  u16  sid;           // Status ID  { 0, 4, 8, or 12 }
  u32  flag;          // Status flag
  u32  mask;          // Status mask
} uObjTxtrBlock_t;   // 24 bytes
```

2. Texture load structure `uObjTxtrTile_t` for using `LoadTile`

```
typedef      struct {
  u32  type;          // by type G_OBJLT_TXTRTILE
  u64  *image;        // texture source address on DRAM
  u16  tmem;          // TMEM word address of loading destination (8byteWORD)
  u16  twidth;        // Texture width specified by macro GS_TT_TWIDTH()
  u16  theight;       // Texture height specified by macro GS_TT_THEIGHT()
  u16  sid;           // Status ID  { 0, 4, 8, or 12 }
  u32  flag;          // Status flag
  u32  mask;          // Status mask
} uObjTxtrTile_t;    // 24 bytes
```

3. TLUT load structure `uObjTLUT_t`

```
typedef      struct {
  u32  type;          // by type G_OBJLT_TLUT
  u64  *image;        // texture source address on DRAM
  u16  phead;         // first TLUT area number 256 ≤ phead ≤ 511
  u16  pnum;          // number of TLUT to be loaded - 1
  u16 zero;           // always 0
  u16  sid;           // Status ID  { 0, 4, 8, or 12 }
  u32  flag;          // Status flag
  u32  mask;          // Status mask
} uObjTxtrTLUT_t;    // 24 bytes
```

The shared structure, `uObjTxtr union`

```
typedef union {
  uObjTxtrBlock_t     block;    // texture load parameter using LoadBlock
  uObjTxtrTile_t      tile;     // texture load parameter using LoadTile
  uObjTxtrTLUT_t      tlut;     // TLUT load parameter
  long long int       force_structure_alignment;
} uObjTxtr;
```

gSPObjLoadTxtr

```
gSPObjLoadTxtr(Gfx *gdl, uObjTxtr *tx)
gsSPObjLoadTxtr(uObjTxtr *tx)

Gfx             *gdl;  The display list pointer
uObjTxtr        *tx;   The pointer to the texture load data structure
```

`gSPObjLoadTxtr` performs each loading operation by referring to the texture loading parameters which are held by the above-mentioned three structures. The three structures have the common member variables `type`, `image`, `sid`, `flag`, and `mask`. First, we will explain these five common member variables.

type

`gSPObjLoadTxtr` distinguishes each structure using the value of `type`, the structure's member variable. Each value of `type` and corresponding structure, and each operation is shown below.

`type` Value	Structure	Operation
`G_OBJLT_TXTRBLOCK`	`uObjTxtrBlock_t`	texture load using `LoadBlock`
`G_OBJLT_TXTRTILE`	`uObjTxtrTile_t`	texture load using `LoadTile`
`G_OBJLT_TLUT`	`uObjTLUT_t`	loading of `TLUT`

image

`image`, the member variable, specifies the texture data in the main memory to be loaded, or `TLUT` data address. This texture data must be 8 byte aligned.

sid, flag, and mask

These three member variables are used for bypassing the reloading operation if the texture in question is already loaded. If the requested texture is already loaded, `g[s]SPObjLoadTxtr` will not perform the load operation.

To determine the existence of the texture in question in TMEM using the RSP, the RSP must analyze the loading destination area for each texture load operation. This is time consuming, and not a very good option.

In S2DEX, the loading destination area data are included in texture data structure. Therefore, rather than performing analysis using the RSP, simple calculation will determine whether or not the loading operation needs to be performed.

For example, when texture data are loaded to TMEM, an ID which corresponds to the loaded texture can be written to a status area. By simply comparing the IDs when the next TMEM loading operation is performed, the loading question can be resolved rather easily.

The loading decision method used by S2DEX is an extension of the above concept. When partial loading by dividing TMEM is performed, S2DEX can also make loading decisions for different parts of TMEM using two 32 bit variables (`flag` and `mask`); this makes partial loading possible.

The RSP provides four 32 bit status variables in the status region. When the microcode starts up, these variables are set to 0. `sid` will determine which status value to use. `sid` can assign one of the values {0, 4, 8, 12}.

`g[s]ObjLoadTxtr` actually makes the loading decision using the steps below.

1. Check the condition of (`Status[sid] & mask == flag`).
2. If the result is true, assume that the texture is already loaded and terminate the loading operation.
3. If the result is false, load the texture, and change Status[sid] to:

```
Status[sid] = (Status[sid] & ~mask) | (flag & mask);
```

The easiest way to use `flag` is to assign -1 (=0xffffffff) to `mask`, and texture's source data address (= the value of the member variable "`image`") to `flag`. If there is no texture data starting from the same address, this will act as a texture cache.

Also, when `(flag & ~mask) != 0`, the condition will always be false, and texture will always be loaded.

The next example will divide TMEM into two areas and control each area. Here, assign Status[0]'s bits 31~16 to the first half of TMEM, and assign bits 15~0 to the last half of TMEM. Assign the sequence number to each texture. The value of `sid` is always 0.

	Load Area	flag	mask
A: texture 1	0 to 255	0x00010000	0xffff0000
B: texture 2	256 to 511	0x00000002	0x0000ffff
C: texture 3	0 to 511	0x00030003	0xffffffff
D: texture 3 only the last half	256 to 511	0x00000003	0x0000ffff

At C, the entire texture 3 is loaded. Even though the loading operation of A changes the first half, since the TMEM's last half retains texture 3 data. The request for loading texture 3 at D to the last half will not require actual loading.

Similar to this example, S2DEX has GBI `gSPSelectDL` / `gSPSelectBranchDL`, which performs a DL branching operation, using the same principle as the operation using `status`.

The member variables of other the structures are explained in the following paragraphs.

1. Texture load using LoadBlock (`uObjTxtrBlock_t` structure)

tmem

The texture's loading destination TMEM address is assigned to `tmem` in DoubleWord units. Normally, this loading address is used as the value of `imageAdrs` of `uObjSprite` structure. If this value is to be specified in pixel units, the macro `GS_PIX2TMEM()`, described earlier, will become useful.

tsize

The size information of the texture to be loaded is assigned to `tsize`. To obtain this value from texture size, the macro `GB_TB_TSIZE()` is used.

```
 GS_TB_TSIZE(pix,siz): tsize setting
 pix:        the number of texels to be loaded (=width of texture X height of
texture)
 siz:        1 texel size, specify G_IM_SIZ_*
```

tline

The width information of the texture to be loaded is assigned to `tline`. Use the macro `GS_TB_TLINE()` for obtaining the value from the texture width.

```
GS_TB_TLINE(pix,siz): setting of tline
pix:             the number of texel of texture width
siz:             1 texel size, specified by G_IM_SIZ_*
```

2. Texture load by `LoadTile` (`uObjTxtrTile_t` structure)

tmem

This member variable is common to the load operations using `LoadBlock`. The TMEM texture load destination address is assigned to `tmem` in DoubleWord units.

twidth

The load texture width information is assigned to `twidth`. Use the macro `GS_TT_TWIDTH()` to obtain the value from texture width.

```
GS_TT_TWIDTH(pix,siz): setting of twidth
pix:             texture width
siz:             1 texel size specified by G_IM_SIZ_*
```

theight

The height information of the texture to be loaded is assigned to `theight`. Use the macro `GS_TT_THEIGHT()` to obtain the value from texture height.

```
GS_TT_THEIGHT(pix,siz): setting of theight
pix:             texture height
siz:             1 texel size, specified by G_IM_SIZ_*
```

3. TLUT load (`uObjTLUT_t` structure)

phead

The first TLUT area number is assigned to `phead`. The palette number can be obtained by adding 256 to the normal palette ID. Therefore, the value ranges from 256 to 511. Use the `GS_PAL_HEAD()` macro for this setting.

```
GS_PAL_HEAD(head): setting of phead (add 256 to head)
head: first ID of TLUT to be loaded
```

pnum

A value representing "(the-number-of-colors-of-the-loaded-TLUT) -1" is assigned to `pnum`. Use the `GS_PAL_NUM()` macro for this setting.

```
GS_PAL_NUM(num): setting of pnum (num -1)
num: the number of TLUT to be loaded
```

zero

This member is not used in `uObjTLUT_t`. However, to maintain compatibility with other structures, always assign 0 to `zero`.

The following illustrates an example of the set-up for the three structures.

1. RGBA16 Texture load using `LoadBlock`

```
uObjTxtr objTxtrBlock_RGBA16 = {
  G_OBJLT_TXTRBLOCK,                         /* type    */
  (u64 *)textureRGBA16,                      /* image   */
  GS_PIX2TMEM(0,      G_IM_SIZ_16b),         /* tmem    */
  GS_TB_TSIZE(32*32, G_IM_SIZ_16b),          /* tsize   */
  GS_TB_TLINE(32,     G_IM_SIZ_16b),         /* tline   */
  0,                                         /* sid     */
  (u32)textureRGBA16,                        /* flag    */
  -1                                         /* mask    */
};
```

2. CI4 Texture load using `LoadTile`

```
uObjTxtr objTxtrTile_CI4 = {
  G_OBJLT_TXTRTILE,                          /* type    */
  (u64 *)textureCI4,                         /* image   */
  GS_PIX2TMEM  (0,  G_IM_SIZ_4b),           /* tmem    */
  GS_TT_TWIDTH (32, G_IM_SIZ_4b),           /* twidth  */
  GS_TT_THEIGHT(32, G_IM_SIZ_4b),           /* theight */
  0,                                         /* sid     */
  (u32)textureCI4,                           /* flag    */
  -1                                         /* mask    */
};
```

3. TLUT load

```
uObjTxtr objTLUT_CI4 = {
  G_OBJLT_TLUT,                              /* type    */
  (u64 *)textureCI4pal,                      /* image   */
  GS_PAL_HEAD(0),                            /* phead   */
  GS_PAL_NUM(16),                            /* pnum    */
  0,                                         /* zero    */
  0,                                         /* sid     */
  (u32)textureCI4pal,                        /* flag    */
  -1                                         /* mask    */
};
```

Compound Processing GBI

In actual game development, combining the Texture Load GBI and the sprite Drawing GBI is sometimes advantageous for controlling Sprites. S2DEX provides the mechanism to control the two GBIs with one GBI. The following is an explanation of compound processing of the GBIs.

`uObjTxSprite` Structure

`uObjTxSprite` structure, which is shown below, has been constructed by combining `uObjTxtr` structure and `uObjSprite` structure. The pointer to `uObjTxSprite` structure is provided to the compound processing GBI as the parameter.

```
typedef struct  {
  uObjTxtr      txtr;
  uObjSprite    sprite;
} uObjTxSprite;          /* 48 bytes */
```

gSPObjLoadTxRect

```
gSPObjLoadTxRect(Gfx *gdl, uObjTxSprite *txsp)
gsSPObjLoadTxRect(uObjTxSprite *txsp)

Gfx          *gdl; display list pointer
UObjTxSprite *txsp; The pointer to texture load and sprite draw data structure
```

The `g[s]SPObjLoadTxRect` GBI performs the Texture Load operation, and then draws a non-rotating sprite.

Essentially, this command performs two GBI operations `g[s]SPObjLoadTxtr` and `g[s]SPObjRectangle` with one GBI. The results of (A) and (B) shown below are identical.

```
(A) gsSPObjLoadTxRect(txsp);

(B) gsSPObjLoadTxtr(&(txsp->txtr));
    gsSPObjRectangle(&(txsp->sprite));
```

gSPObjLoadTxRectR

```
   gSPObjLoadTxRectR(Gfx *gdl, uObjTxSprite *txsp)
   gsSPObjLoadTxRectR(uObjTxSprite *txsp)

   Gfx               *gdl;  The display list pointer
   uObjTxSprite      *txsp; The pointer to the texture load and the sprite drawing
data
                            structure
```

The g[s]SPObjLoadTxRectR GBI performs the Texture Load operation, and then draws a non-rotating sprite referencing a 2D matrix.

Essentially, this command performs two GBI operations, `g[s]SPObjLoadTxtr` and `g[s]SPObjRectangleR` with one GBI. The results of (A) and (B) shown below are identical.

```
(A) gsSPObjLoadTxRectR(txsp);

(B) gsSPObjLoadTxtr(&(txsp->txtr));
    gsSPObjRectangleR(&(txsp->sprite));
```

gSPObjLoadTxSprite

```
   gSPObjLoadTxSprite(Gfx *gdl, uObjTxSprite *txsp)
   gsSPObjLoadTxSprite(uObjTxSprite *txsp)

   Gfx               *gdl;  The display list pointer
   uObjTxSprite      *txsp; The pointer to the texture load and the sprite drawing
data
                            structure
```

The `g[s]SPObjLoadTxSprite` GBI performs the Texture Load operation, and then draws a rotating sprite.

Essentially, this command performs two GBI operations, `g[s]SPObjLoadTxtr` and `g[s]SPObjSprite` with one GBI. The results of (A) and (B) shown below are identical.

```
(A) gsSPObjLoadTxSprite(txsp);

(B) gsSPObjLoadTxtr(&(txsp->txtr));
    gsSPObjSprite(&(txsp->sprite));
```

Conditional Branching GBI

We have explained that S2DEX is using the RSP's Status for making a loading decision. Here, we will explain the GBI which uses Status for DL branching and linking.

gSPSetStatus

```
   gSPSetStatus(Gfx *gdl, u8 sid, u32 val)
   gsSPSetStatus(u8 sid, u32 val)

   Gfx               *gdl;  display list pointer
   u8                 sid;  Status ID { 0, 4, 8, or 12 }
   u32                val;  A value the user desires to set
```

`g[s]SPSetStatus` assigns the value of `val` to the Status area (`Status[sid]`) specified by `sid`. The Status value is referenced for Texture Loading and making conditional branching decisions.

gSPSelectDL

```
gSPSelectDL(Gfx *gdl, Gfx *ldl, u8 sid, u32 flag, u32 mask)
gsSPSelectDL(Gfx *ldl, u8 sid, u32 flag, u32 mask)

Gfx             *gdl;  display list pointer
Gfx             *ldl;  display list to be linked
u8               sid;  Status ID { 0, 4, 8, or 12 }
u32              flag; Status flag
u32              mask; Status mask
```

`g[s]SPSelectDL` inspects `Status[sid]` using the same method used for texture load decision making. Depending on the True/False result, other display lists are called.

`g[s]SPSelectDL` determines whether or not to call the display list by going through the following steps.

- Check the condition of `(Status[sid] & mask) == flag`
- If the result is true, finish GBI without doing anything.
- If the result is false, change the `Status[sid]` by performing:

  ```
  Status[sid] = (Status[sid] & ~mask) | (flag & mask);
  ```

  and call display list "`ldl`".

gSPSelectBranchDL

```
gSPSelectBranchDL(Gfx *gdl, Gfx *bdl, u8 sid, u32 flag, u32 mask)
gsSPSelectBranchDL(Gfx *bdl, u8 sid, u32 flag, u32 mask)

Gfx             *gdl;  display list pointer
Gfx             *link; display list to be linked
u8               sid;  Status ID { 0, 4, 8, or 12 }
u32              flag; Status flag
u32              mask; Status mask
```

`g[s]SPSelectBranchDL` examines `Status[sid]` using the same method used for texture load decision making, and depending on the True/False result branches out to other display lists.

`g[s]SPSelectBranchDL` determines whether or not to call the display list using the following steps.

- Check the condition of `(Status[sid] & mask) == flag`
- If the result is true, finish GBI without doing anything.
- If the result is false, change the `Status[sid]` by performing:

  ```
  Status[sid] = (Status[sid] & ~mask) | (flag & mask);
  ```

  and branch out to display list "`ldl`".

Chapter 4 Emulation Functions

These are functions for using the CPU to emulate S2DEX GBI functions.

guS2DEmuGBgRect1Cyc

```
void        guS2DEmuGBgRect1Cyc (Gfx **gdl_p, uObjBg *bg);
```

This function uses the CPU to emulate the action of S2DEX function `gSPBgRect1Cyc` by combining other GBI's.

Parameters:	gdl_p	Pointer to pointer to display list * The value for gdl_p is automatically calculated.
	bg	Pointer to uObjBg structure

Calling `gSPBgRect1Cyc (gdl ++, bg)` can be replaced by `guS2DEmuGBgRect1Cyc (&gdl, bg)`. Refer to "`gSPBgRect1Cyc`" on page 13 for an explanation of the parameter `bg`.

In addition, in order to notify the main routine that a scissoring box setting and Texture Filter setting, the function `guS2DEmuSetScissor`, discussed below, must be called before `guS2DEmuBgRect1Cyc`.

This function produces GBI's which are functional not only in S2DEX, but in the F3DEX series as well. Because of this, one microcode can be processed when displaying a scaled scrolling BG screen and a 3D model at the same time.

guS2DEmuSetScissor

```
void  guS2DEmuSetScissor (u32 ulx, u32 uly, u32 lrx, u32 lry, u8 bilerp);
```

This function sets the scissoring parameters and Texture Filter referred when the function `guS2DEmuBgRect1Cyc` is processed.

Parameters:	ulx	upper left X coordinate of scissor box (u10.0)
	uly	upper left Y coordinate of scissor box (u10.0)
	lrx	lower right X coordinate of scissor box (u10.0)
	lry	lower right Y coordinate of scissor box (u10.0)
	bilerp	set to value other than 0 to perform Bilerp interpolation processing on the image, or set to 0 for PointSample.

Normally, the range of the scissor box set by `g[s]DPSetScissor` is handled by this function as parameters. In addition, the initial values for `ulx, uly, lrx, lry,` and `bilerp` are 0, 0, 320, 340, 0, respectively, which are settings that draw to a 320x240 pixel frame buffer with PointSample.

This function only needs to be called once before `guS2DEmuBgRect1Cyc` is called. As long as there is no change in the scissor box and Texture Filter, it only needs to be called once during game initialization, and doesn't need to be called every time a frame is drawn.

Chapter 5 DEBUG Information Output Function

There are 2 versions of S2DEX Microcode; one version for debugging and another version for release. The relationship between the two microcodes is the same as the relationship between libultra_rom.a and libultra_d.a.

Although the debug version microcode, `S2DEX_D` is slower than the release version of the microcode, it has the following additional features.

- Outputs the display list processing log.
- In the event of bad input or encountering undefined commands, stops RSP and reports the problem to the CPU.

Investigation of problems, such as finding the cause of a runaway RSP, will become easier by checking the display list processing log.

To use `S2DEX_D`, it is necessary to prepare an output buffer for the RSP display list processing log. The size must be the same as the display list, and must be 8 byte aligned.

Once the area is reserved, provide the pointer data of the first address of the area to `data_size`, which is a member variable of the `OSTask` structure. This member variable is not used in the S2DEX and F3DEX series to mean the size of the DL is the essential meaning. A remnant of N64 OS/Library Version 1.0, it is used as a log output buffer.

This address must not be the Segment address. When `gspS2DEX.fifo_d.o` activates as microcode, it is stored in the address specified by the process log.

For details concerning the processing log's display methods, please refer to the function `ucDebugGfxLogPrint()` in the sample program `uc_assert.c`. Also, for details concerning the decision making process for stopping the RSP, please refer to `ucCheckAssert()` in the same file.

Note: **The OSTask structure's member variable `yield_data_size` was used to set the log buffer in S2DEX Release 0.75 and older, but this was switched to `data_size` in Release 0.76 and later. Please note that the display function `ucDebugGfxLogPrint()` has also been corrected.**

Chapter 6 Installation of S2DEX Package

The description here applies to S2DEX Microcode when it is received as a patch. If the package is included in the N64 OS/Library that you received, the work described here is not necessary.

S2DEX Microcode consists of the following files:

`gspS2DEX.fifo.o`	S2DEX Microcode
`gspS2DEX.fifo_d.o`	S2DEX Microcode (for Debugging)
`include/gs2dex.h`	Include files for S2DEX
`libultra/Makefile`	Makefile for updating libultra
`libultra/us2dex.o`	Initialization routine for BG structure
`libultra/us2dex_emu.o`	Scaleable BG drawing routine
`sample/*`	S2DEX Sample programs

`libultra*.a` are created by executing the make command in the `libultra` directory. Copy `libultra*.a` files to `/usr/lib`. Also, copy `gspS2DEX.fifo.o` and `gspS2DEX.fifo_d.o` to `/usr/lib/PR`, and copy `include/gs2dex.h` to `/usr/include/PR`.

In addition, perl is necessary to compile affiliated sample programs. Please install the following packages from the IRIX 5.3/6.x CD.

For IRIX 5.3:
    eoe2.sw.gifts_perl

For IRIX 6.x:
    eoe2.sw.gifts_perl